Marko Ketonen

Supporting Configuration Modeling in an Evolving Software Component Environment

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Software Business and Engineering

Author and name of the thesis:
Marko Ketonen:
Supporting Configuration Modeling in an Evolving Software Component Environment
Date: 24.2.2004 **Number of pages:** 98 + 7
Department: Computer Science
Professorship: T-76
Supervisor:
Professor Casper Lassenius
Instructor:
M.Sc. Lauri Vuornos, Smartner Information Systems Ltd.

This thesis aims to resolve a number of software product configuration related problems in a case company, Smartner Information Systems Ltd. The purpose is to find a way to express product configuration information using a product configuration modeling technique in such a way that the problem scenarios can be solved.

The problem field in the case company is expressed through a set of product configuration scenarios where the problems exist. Each of these scenarios introduce an actor or actors whose daily work these problems affect. From the scenarios, a set of requirements are derived to act as evaluation criteria for a set of existing product configuration modeling techniques.

The evaluation of product configuration modeling techniques aims to find a set of good practices to use in the case company's problem field. These practices are combined to create the 'ideal' modeling technique for the case company.

In the next phase of the thesis, the 'ideal' modeling technique is implemented in a way that the actors of the scenarios can use the implementation to solve the problems existing in the scenarios. The implementation is then evaluated against the scenarios and requirements derived from them.

Keywords: software product configuration, configuration modeling, configuration modeling technique

Tekijä ja työn nimi: Marko Ketonen: Supporting Configuration Modeling in an Evolving Software Component Environment	
Päivämäärä: 24.2.2004	**Sivumäärä:** 98 + 7
Osasto: Tietotekniikan osasto	**Professuuri:** T-76
Työn valvoja: Professori Casper Lassenius **Työn ohjaaja:** DI Lauri Vuornos, Smartner Information Systems Oy	

Tämän diplomityön tavoitteena on ratkaista joukko ohjelmistotuotteen konfigurointiin liittyviä ongelmia case-yrityksessä, Smartner Information Systems Oy:ssä. Tarkoituksena on löytää tapa, jolla tuotekonfigurointiin liittyvää tietoa voitaisiin ilmaista tavalla, joka toimisi ratkaisuna ongelmille.

Case-yrityksen ongelmakenttä kuvataan skenaarioiden avulla, jotka kuvaavat miten ongelmat esiintyvät. Jokaisella skenaariolla on toimija tai toimijoita, jotka ovat yrityksen työntekijöitä, joiden päivittäiseen työhön ongelmat vaikuttavat. Näistä skenaarioista johdetaan joukko vaatimuksia, joiden pohjalta arvostellaan joukkoa olemassaolevia ohjelmistotuotekonfiguroinnin mallinnustekniikoita.

Tuotekonfigurointitapojen arvioinnilla pyritään löytämään joukko hyviä toimintamalleja, joita voidaan käyttää case-yrityksen ongelmien ratkaisemiseksi. Nämä käytännöt yhdistetään 'ideaaliseksi' mallinnustekniikaksi case-yritystä varten.

Työn seuraavassa vaiheessa 'ideaalinen' mallinnustekniikka toteutetaan siten, että skenaarioiden toimijat voivat käyttää toteutusta ongelmien ratkaisemiseksi. Lopuksi toteutusta arvioidaan skenaarioiden ja niistä johdettujen vaatimusten pohjalta.

Avainsanat: ohjelmistotuotteen konfigurointi, konfiguraation mallintaminen, konfiguraation mallinnustekniikka

Preface

This thesis has been done for Smartner Information Systems, Ltd. I would like to thank the company for giving me the resources to do this thesis.

I would like to thank my instructor M.Sc. Lauri Vuornos for his determination in seeing my thesis finished and me graduated and also all of the time and expertise he has devoted into the process.

I also thank my supervisor, Professor Casper Lassenius for all the feedback I have received from the sessions that we have had. I also appreciate the positive attitude and encouragement he has given me during the writing process.

I issue a special thank you to Dr. Tech. Tomi Männistö for the help he has given me in all aspects and phases of the work. I propably would have not been able to make it without his feedback and expertise on the subject.

Thanks goes also to Terhi Norokorpi for giving me most valuable lessons in English grammar.

I thank my family for giving me the basis of making it this far and for the support I have received through my whole life. I thank my friends for giving me an alternative in spending my time besides writing. A special thanks goes to my friends at HUT, especially the ones I have been competing to graduate with. Without the continuous chat about thesises it would have been too fruitless a task to write.

Helsinki, 24.2.2004

Marko Ketonen

Contents

Terms and Abbreviations

ADL Architecture Description Language, a common name for all languages capable of describing any architectural structure.

CBR Case Based Reasoning, a method where past experiences are stored into a knowledge base and used when solving new problems.

CVS Concurrent Versions System, a system capable of storing different versions of files, enables concurrent development using the same file repository.

Exchange A personal information management server from Microsoft.

HTML HyperText Markup Language, describes a language which can be used to create documents viewable by web browsers.

PDA Personal Digital Assistant, any hand-held device, which provides the user some level of computing, especially for personal and business use, like contacts, calendar etc.

PDM Product Data Management, represents all actions done to maintain information about different product data.

PHP a simple scripting language that can be used to create small and simple additional functionality (like accessing shell commands) to WWW pages.

PIM Personal Information Management, managing personal information like emails, calendar, contacts etc.

SyncML A protocol used for synchronizing different kinds of data (calendar, contacts, etc.) between a device and a server. Also known as OMA data synchronization.

UML Unified Modeling Language, a box-and-pointer type of modeling language used for e.g. architectural design.

WAP Wireless Application Protocol, a protocol used to transfer content into devices with small bandwidth capabilities like mobile phones.

WWW World Wide Web, an Internet extension where content can be stored and displayed visually using web browsers.

XML eXtensible Markup Language, a structured language which can be used to describe any structured data.

XSL eXtensible Stylesheet Language, can be used to transform an XML document into some other format (like HTML). Contains processing instructions to different XML elements and attributes.

Chapter 1

Introduction

This chapter introduces the background for the study. It also presents the research problem, goals and objectives, scope, methodology and the structure of the paper.

1.1 Background

The software industry has not been around for long, so the working methods (processes, design, implementation etc.) are still quite undeveloped when compared to traditional branches of industry. Because of this, many things are done in ad hoc fashion in software companies. A lot of research effort has been directed into standardizing ways of working. New working ways have been introduced, but none of these have really established a de facto status in the industry.

Mobile software industry is a part of the whole software industry and one of the youngest ones so the working methods are even more premature that in traditional software industry. It introduces an environment where change is a common phenomena. New technologies emerge in the form of new end user devices, mobile networks etc. One must concentrate on coping with change, but as the companies are usually quite small, the working methods also need to be lightweight, so no additional organization is needed to operate them.

A mobile software product has to operate in a complex environment of databases, servers and end user mobile devices. The limited resources of a mobile software company lead to the fact that not all can be done by the company itself. The company needs to concentrate on their core product as other needed functionality is acquired using subcontracting, open source software and purchasing commercially available software. The combination of these makes the fully functional mobile software product which is sold to the customer. The working method that concentrates on building the product from these high level components is called product configuration.

In addition to combining a mobile software product with its environment, the internal structure of the product needs to be considered as well. Modern software

product development is usually component oriented, i.e. the product is divided into functional components, which implement a restricted part of the whole product functionality. Changes made to individual components can have an effect on related components and these effects need to be taken into account when the product is constructed from the components.

In order to effectively control and manage the way the product is constructed from the internal and external components, a way to model the relationships between the components is needed. In traditional industry such as building machinery, this has been studied a lot. But whereas a piece of machinery is constructed from parts whose compatibility is visible to the human eye (one part does not simply connect to the other one), a software product is more abstract and the relationships between the components is not visible, but needs to be recorded somehow to form an understanding of the compatibility between the components.

1.1.1 Research Background

This study aims to find a suitable way to support modeling of software component configurations in a mobile software company. To build an emphasis on actual implementation instead of focusing on theoretical level, the study is conducted for a case company, where there is a need for developing practical ways to model configurations. The case company and its product offering is described in short here.

Case Company

The case company is Smartner Information Systems, which is a small mobile software company empowering business mobility solutions. The company was founded in 1999. It has offices in Helsinki Finland, Cambridge UK and Paris France. The company currently employs about 35 people working in product development, customer services and support, product management, sales and marketing and administration.

Case Company's Product Offering

Smartner offers its world-class mobile technology competence for operators and service providers who need tools for building mobile services for enterprises. With the Smartner solution, this can be done quickly, efficiently and intelligently. The Smartner solution, Smartner Duality, is comprised of mobile applications that are offered to enterprises as hosted services, and technology modules enabling service management, data security and connectivity between the operator and enterprises. Using the categorization of software products defined in (Detlev et al., 1999) the Smartner solution can be described as enterprise software.

The solution can be roughly divided into three categories: *Push*, *Synchroniza-*

tion and *Browsing*. The *Browsing* side of the solution allows its users to access their PIM data using popular access methods like WAP and WWW depending on the capabilities of the devices used (e.g. laptop computer, PDA, mobile phone). The *Synchronization* part means keeping device PIM data up-to-date with the data on the server i.e. what the user can see using the desktop PIM applications (e.g. calendar application) using little or no user interaction. The *Push* side means keeping the device PIM data always up-to-date by pushing new PIM data and changes to existing data to the device without any user interaction. In *Push*, everything is done underneath the usual device usage, meaning that everything is intergrated into the native PIM applications already in the device. In *Synchronization*, the user can use a separate SyncML client in the device to synchronize PIM data whenever needed.

1.2 Research Problem

The problem being solved for the case company can be described using a set of scenarios where the problem is visible. In the context of this study, a scenario means an actual event existing in the day-to-day activities performed by the company employees. The scenario is linked to some company function (sales, support, maintenance etc.) and a company employee working in this function. The employee will be called the scenario actor in future reference in this study. One definition of a scenario can be the use case (Fowler and Scott, 2000) which is used to describe the requirements of software. The definition of a use case is similar to the one used here for scenarios, but the usage is different so the word "scenario" is used. These scenarios will be used as a basis for defining the research problem for the study as well as defining the goals and objectives. The following section describes the scenarios.

To understand the scenarios, the reader should be familiar with the domain and the concepts used there. The following describes the concepts used in the domain and their relationships.

- **A solution** is something that is delivered to a customer. In terms of configurations, it is the initial configuration installed at the customer premises consisting of the products of the company and its operational environment. A solution is something that is sold to satisfy a customer need by providing a service or services. If the products inside the solution offer any level of customization, the customized versions of the products are part of the solution.
- **Environment** is where a solution is operated and used. Basically it means everything that is somehow linked with the operation of the solution. This covers hardware (servers, user devices, networks etc.) and software (operating systems, databases, application servers etc.) which are not parts of the products of the company. The line between products and environment is not strict because any part of the environment can also be a part of a product of the company (e.g. a user device and its software) and vice versa (e.g. part-

nering with a client software manufacturer to concentrate to server software development).

- **A service** is something that can be offered to a customer. A solution implements a way to provide a service or services to customers. A service can be considered as an interface between a customer and a solution.
- **A product** is something that a solution consists of. A product is a set of components packaged to form an entity which can provide some kind of service. In addition to software components, a product usually consists of all the documentation related to it and possibly some hardware components (like servers, client devices etc.). Products are the part of a solution that a company creates as environment is the part already existing.
- **A component** can have many abstraction levels from a mobile device with its embedded software to a single compiled class file. Usually it is considered to be an entity providing some kind of functionality. A software component consists only of software. Components can either be product-specific (when containing only functionality which covers the scope of a single product) or general (when containing more generic functionality like a library of utilities used in many products).
- **Documentation** means technical documentation, user manuals, environment descriptions and any other written documents about any parts of the domain.

The relationships of the domain concepts is illustrated in figure 1.1.

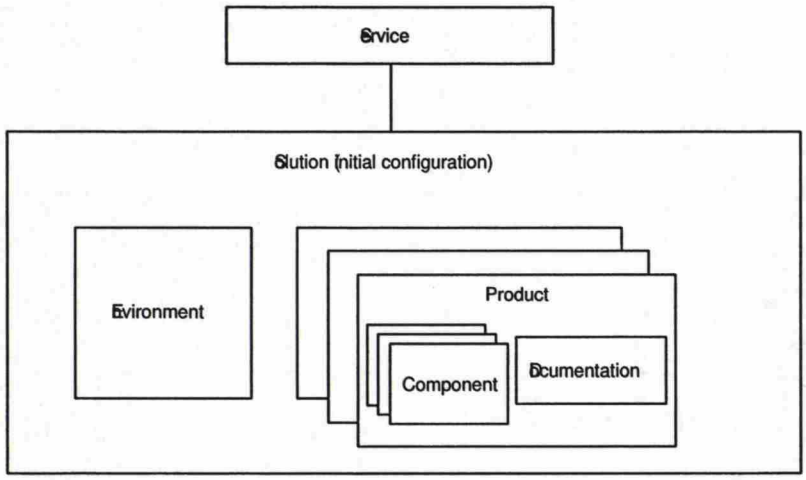

Figure 1.1: Illustration of the domain concepts

A solution consists of an environment and the products created by the company. It implements a service which can be used by the customers. The products operate

in the environment. A product consists of components (hardware or software) and all product-related documentation.

1.2.1 Scenarios

The scenarios will be described by first presenting a story about an actual event happening for a case company employee. After that the scenario contents and the actual problem will be described in further detail and the current way to handle the event is also described. The scenarios are based on interviewing the employees working on different areas in the case company and on the working experience of the author.

Scenario 1: Reconfiguration

A system integrator is in a customer country where a new version 3.0 of the Smartner solution was installed a couple of months ago. The customer has been testing the new version in an integration environment and has reported a number of different issues, which need to be resolved before the delivery can be accepted. The system integrator's goal is to install the latest patch versions for the 3.0 version that should resolve most of the issues reported. For some components, there is a clear patch version available, which can be upgraded by simply replacing the old component with the patch component but for some components there are new versions of the component available and no clear indication how the two versions differ in terms of interoperability with other components. Now the system integrator needs to decide whether to reinstall the whole configuration already existing in the integration environment or to install only the needed components to make the changes apply.

Reconfiguration means that an existing component configuration is being upgraded by replacing only some components, not all of them. An existing configuration consists of the internal and external software components already installed. It also consists of all installation-specific customization (configuration [1] files, customer-specific UI layouts etc.), which need to be preserved to be used with the upgraded version as well. The amount of installation-specific customization is usually the main reason why a configuration cannot be reconfigured as a whole. As new versions of a product are released, existing configurations are likely to be upgraded into the new version at some point. There is also a need to make so called patch upgrades to existing configurations when there are bug fixes or new product features available. The actor in the case of reconfiguration is usually a system integrator, which is either a company internal employee aware of the product or an external employee using e.g. an API provided by the company. Both of these actors need to be considered in the case of reconfiguration.

Special cases of reconfiguration are the initial installation where the existing

[1] Here, configuration means actual configuration data like IP addresses, log file locations etc.

configuration consists of the third party software components already installed in the installation environment and a case where reconfiguration is done by replacing the existing configuration with a new one. In the case of the initial installation, the third party software components consists of components which there is some control of (e.g. application servers and databases, the supported versions are specified and these versions are required to be installed in the environment) and components for which there is no control of (e.g. external data storage systems, where some information is retrieved).

The current situation in the case company is that there is only a limited amount of information available about the interoperability of different versions of related components. Usually the information is known by the person who has made the change which affects component interoperability, but this is not always the case. For some components, there is a clear patch procedure, where bug fixes are made to a certain version of a component by branching the sources from the version control system. This means that the patch version does not contain any other changes made to the component which can affect its interoperability with other components. A new release is always a snapshot of the latest component versions available. If it is to be used to reconfigure an existing configuration by only upgrading parts of the components, there is no exact information available which components need to be upgraded. Thus the upgrade must be done by gathering information from different parties (developers, system integrators with previous installation experience etc.). This ad hoc way of handling reconfiguration causes lots of extra work for a large group of people and the validity of the newly created configuration can only be verified after it has been used.

Scenario 2: Supportability and Maintainability

A support on-call person answers the support phone. The caller is a high-level executive of a customer company and he is very angry. He says that the solution delivered to them does not work at all. After the on-call person manages to calm the customer representative down, he finds out that there is a couple of level two bugs (so called 'show-stoppers') preventing a number of users from using the service. He remembers that recently there was some talk about fixing bugs related to the ones mentioned by the customer representative. He is not entirely sure whether the bugs have actually been fixed, so he ends the call by saying that the problems are caused by a couple of bugs which are being fixed and will be delivered in the next patch version. The customer representative is not entirely satisfied and mentions that a competitor solution is being trialled and seems to be working better than the solution delivered by the case company.

As the reconfiguration scenario is mainly related to actual actions being made, this scenario is more focused on communication and distributing knowledge about product configurations. When support is offered to an existing configuration, there must be information available about existing bug fixes to the configuration and

possible new features. The configuration model must be able to provide information about differences between existing configurations and available configurations.

This scenario is closely linked with the reconfiguration scenario as the support requests made might lead to reconfiguration of the existing configuration.

Currently in the case company there are parts of the product where delivering bug fixes to customers is quite clearly defined through delivering patches. This method of operations is not applied to all components which may cause problems in the future. For the components, where such patch delivery system does not exist, the bug fixes are delivered by upgrading old versions with new versions of the components. Due to this, the new version usually contains other changes than the required bug fix and the other changes might affect the interoperability of the component with the existing configuration.

Scenario 3: The Sales Perspective

A sales director is meeting high level executives of a potential customer company. After a long discussion about pricing the conversation shifts into describing the customer environment and the new devices in the market, which they want the solution they buy to support. "Our market research shows that the new Microsoft Smartphone will be taking a large portion of the PDA market during the next quarter. How does your solution support this device? And also it seems that most of our business customers are upgrading their systems to Exchange 2003. How do you support it?". The sales director does not exactly know whether these end use devices and office solutions are supported, she/he has to circumvent the questions by saying that she/he will ask the product person back in the office how these are supported and will get back to the issue in an email or in the next meeting.

When a solution is being sold to a customer, the sales person needs to have enough information about the case company's products in order to sell the right solution to the customer. This can be described as building a configuration to satisfy customer needs. This is specially the case when the whole product offering is not always delivered, only parts of it. Acquiring knowledge about the environment where the solution is installed is important here as it helps the system integrator to make the initial installation (a special case of the reconfiguration scenario).

To this point, the whole product offering has always been delivered to a customer as a solution and thus no extra product variation has not been needed. In the future, there are plans to deliver different variations of the product to different customers which raises a need to manage product variants. There has also been difficulties in mapping the environment supported by the company products (supported end user devices etc.).

Scenario 4: The Future Aspect

A sales director and a product manager are talking about the future with an existing customer. The product manager introduces the case company's roadmap, which introduces new versions of the product with new features to be released in the future. The customer representatives seem very interested in the new features and willing to buy these in the near future. They also have some new ideas concerning new features needed in the market. The conversation leads to starting the planning of an upgrade project aiming to incorporate a set of new features to the solution already delivered to the customer.

After-sales to existing customers is usually an easier way to create revenue than to find new customers (Tomi Männistö, Timo Soininen, Juha Tiihonen and Reijo Sulonen, 1999). It may also be less dependent on market fluctuations and thus provides a more steady source of income. In order to make the resulting reconfiguration task more easier, the planning of after-sales needs to include some level of reconfiguration planning. The reconfiguration task will be much easier, if the upgrade from the existing configuration to the new, still "on the drawing board" like configuration, is planned as a part of the upgrade project. There should be a way to see what the solution looks like after a certain time period. One should be able to link the company roadmap of new product features to future-coming releases.

In the case company, there has recently been a case where a new product release contained a lot of new features, like synchronization of email, calendar and contacts. The upgrade from the previous version of the product was taken into account at some level (database structure upgrade, product documentation listed features added and removed from previous version etc.) but there is still lots of room for improvement.

1.2.2 Defining the Research Problem

Configuration modeling means establishing a view to the component structure of a product. This view can be made by different parties (company members in different tasks, partners, customers etc.) so it needs to be able to support different levels of abstraction and able to give out a variety of information. In addition, the model must provide a mechanism to grasp the issues presented in the scenarios above. The research problem can be stated:

How to support the software component configuration modeling of the case company's solution in a way which is able to respond to the requirements set by the case company's product configuration scenarios?

The solution to the problem can be sought by answering the following questions:

- What is the component composition of the case company's solution?
- What are the requirements set by the scenarios?

- How do existing product configuration modeling techniques respond to the requirements?
- What kind of modeling technique is needed to respond to the requirements?
- What kind of information is needed from the components of the case company's solution?
- How can the modeling technique be implemented in order to be used by the actors in the scenarios?

Question one is needed to give the reader an understanding of the environment where the case company's solution is used. The environment has an effect on requirements set by the scenarios. By answering question two, the requirements for the model are mapped using the information provided by the scenarios. Question three addresses the previous work done in the field and the need to analyze the modeling techniques already developed. As these techniques usually respond to general requirements set by a very high level need, question four addresses the need to create a specific technique which responds to the specific requirements set by the scenarios. Question five looks at the components of the case company's solution in respect of the information needed to create a configuration model of the solution. Question six describes the need to create an actual solution to the problem i.e. an implementation which the actors in the scenarios can use to solve the problem.

These questions also form the structure of the study, which is described in detail in section 1.6. The order in which these questions need to be addressed can be seen from figure 1.2.

1.3 Goals and Objectives

To solve the research problem, one must answer the questions stated in the previous section. The objectives can be set accordingly:

Objective 1 Describe the component composition of the case company's solution.

Objective 2 Determine the requirements for the product configuration model set by the scenarios presented.

Objective 3 Evaluate existing modeling techniques against the requirements.

Objective 4 Define a modeling technique, which is best suited to respond to the requirements.

Objective 5 Identify the information needed from individual product components and their relationships in order to form the needed product configuration model.

Objective 6 Implement the defined modeling technique to be used by the actors in the scenarios to solve the problems presented in the scenarios.

As a result of the study, there should be a good understanding of the component composition of the case company's solution among the users of the model being developed. This means that in addition to the model being created, it should also be used and the users provided with the information they need. The flow from the research problem to the solution through the objectives can be seen from figure 1.2.

In order to somehow measure that the study has given improvement to the situations described in the scenarios, the following goals need to be achieved:

Goal 1 After the study, the system integrator should not need to contact anyone when upgrading an existing configuration. There should be enough information available or the process should be automated to the level that the system integrator can handle the reconfiguration by himself.

Goal 2 The study should reduce the latency in responding to questions about differences between configurations in terms of bug fixes and features. It should also reduce the number of people needed to resolve these issues to the level that by using the configuration modeling implementation, a single person can respond to these questions.

Goal 3 The result of the study should provide the sales representatives of the case company means of describing the Smartner solution to a potential customer in a way understandable to both the sales representative and the customer. It should also give a means of what kind of information is needed before the right solution can be sold to the customer.

Goal 4 The study should provide means of modeling configurations that do not exist i.e. building configurations according to information provided by the company roadmap, customer feature requests etc.

1.4 Research Scope

In terms of Product Data Management, the research focuses only on the product information needed to form configurations out of the individual components. The mechanism to gather e.g. customer and internal test data are outside this research. However, any information needed to tell whether a version of a component works with a version of another component is considered vital in solving the research problem.

The concept of a product feature is considered at a very high level in the study. The product features will be considered at the topic level (e.g. synchronization of contacts and calendar), not going deep into detail about the requirements of a

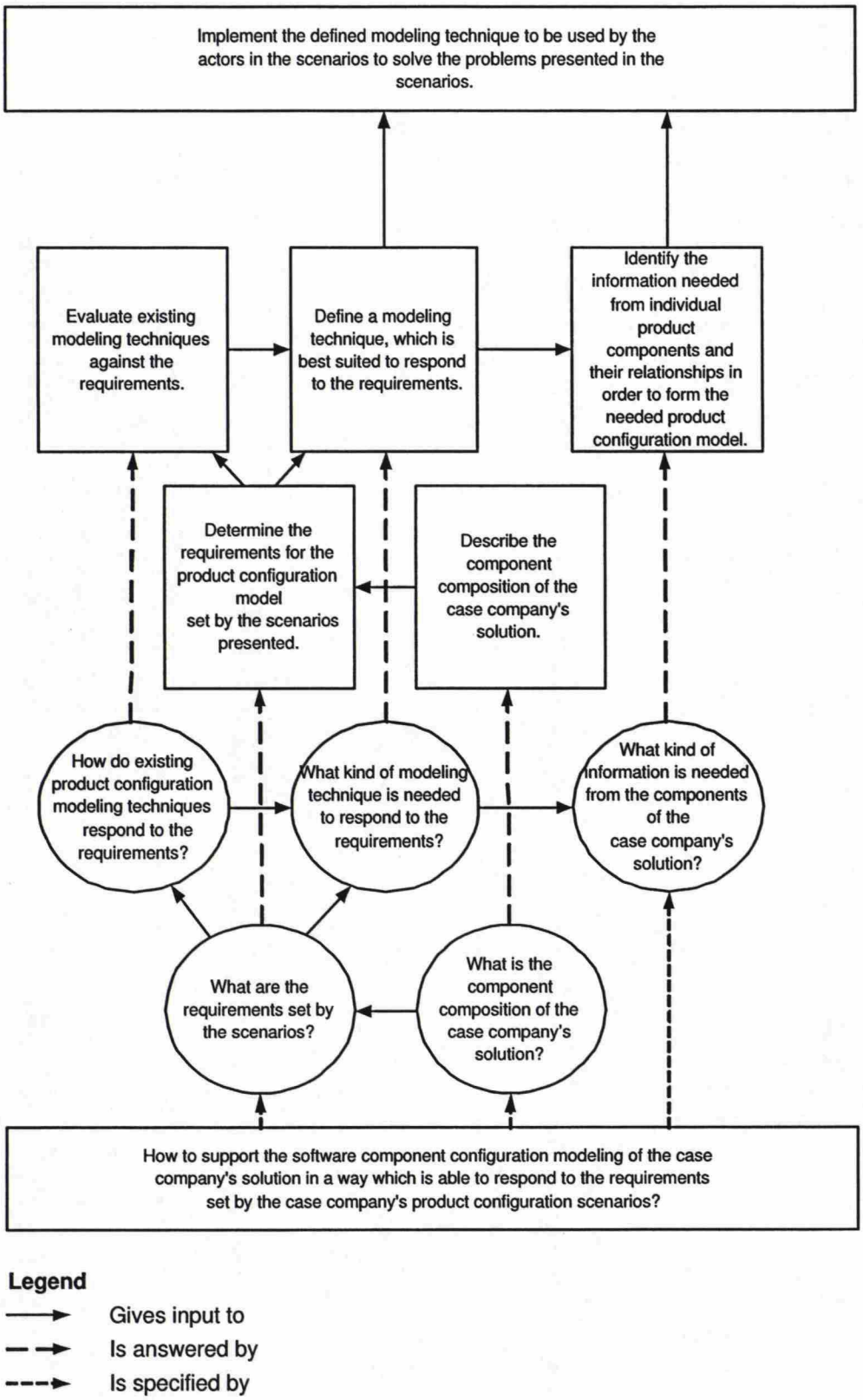

Figure 1.2: Research problem and objectives

feature. Features will be combined into components by mapping which new components and which new versions of existing components are needed to acquire a new feature.

The research aims to find a light-weight way to model configurations which is suitable for small software companies with minimum extra effort to spare in configuration modeling. This requires a high level of automation in all phases of the process in forming a configuration model.

In the context of this research only software components are studied. Hardware components and any documents etc. related to the product or product configuration are not taken into account.

As the main goal of the research is to find a way to model configurations from existing or planned software components, the development phase where components are created (i.e. compiled from source code) cannot be taken into account. Thus the point of view of a developer is limited to having a view to the component composition of the product, not to decide which components to use when building another component.

1.5 Research Methodology

When concerning methodologies, the research consists of three parts. The first part gathers the requirements from the case company's product configuration scenarios. This is conducted by interviewing case company employees working in the positions matching the actors of the scenarios. The author's personal knowledge and expertise is also used. The component composition of the case company's solution is presented for the reader to get an understanding of the effect it has on the scenario requirements.

The second part is conducted as a literature study to gather information about existing techniques for supporting configuration modeling. The study aims to find out the different ways to model configurations and compare them in the context of the research problem and the requirements set by the scenarios.

The third part consists of an implementation to support configuration modeling in the case company. The implementation is done using the selected modeling technique and it should be able to respond to the changes set by the scenario requirements. The implementation is evaluated against the requirements.

1.6 Structure of the Study

The research structure follows the research problem decomposition into questions, goals and objectives presented in figure 1.2:

- **Chapter 2: The Component Composition of Smartner Duality** describes the component composition of Smartner Duality in a relatively high level and describes the environment where it is used.
- **Chapter 3: Scenario Requirements** presents the requirements set by the case company's product configuration scenarios (section 1.2.1).
- **Chapter 4: Existing Configuration Modeling Techniques** analyzes existing configuration modeling techniques against the requirements presented in chapter 2.
- **Chapter 5: The 'Ideal' Modeling Technique** describes the 'ideal' modeling technique based on the analysis of existing modeling techniques and the requirements set by the scenarios.
- **Chapter 6: Information Needed from the Components** presents the information needed from the components to form the ideal product configuration model presented in chapter 4.
- **Chapter 7: Implementation of the 'Ideal' Model** introduces the implementation of the ideal modeling technique.
- **Chapter 8: Evaluation of the Implemented Model** evaluates the implemented model against the requirements from chapter 2.
- **Chapter 9: Conclusions and Future Work** concludes the study and presents any indications of future work.

Chapter 2

The Component Composition of Smartner Duality

This chapter introduces Smartner Duality's component composition. The component composition has an effect on the requirements set by the scenarios, so it is very important to understand the environment where the different components are used. The component composition is presented in figure 2.1.

The two most important interfaces concerning component interoperability are the interfaces between the end user layer and Duality server layer and between Duality server layer and Enterprise Gateway layer. The following sections describe the layers in figure 2.1 and their interoperability between the other layers and the different levels of component abstraction in Smartner Duality.

2.1 The Layers and Their Interfaces

This section describes the layers presented in figure 2.1 and their interfaces to other layers.

2.1.1 End User Layer

The end user layer consists of the end user devices supported by Smartner Duality. In case of the Professional Edition, it also includes software installed on the device. From the environment point of view, the end user layer is the most evolving part of the environment because new end user devices are rapidly brought into the market by vendors and usually the early adopter type of user uses the latest devices in the market. But as there needs to be some idea about interoperability with latest devices, there must always be backward compabitility with the older, but more widely used devices.

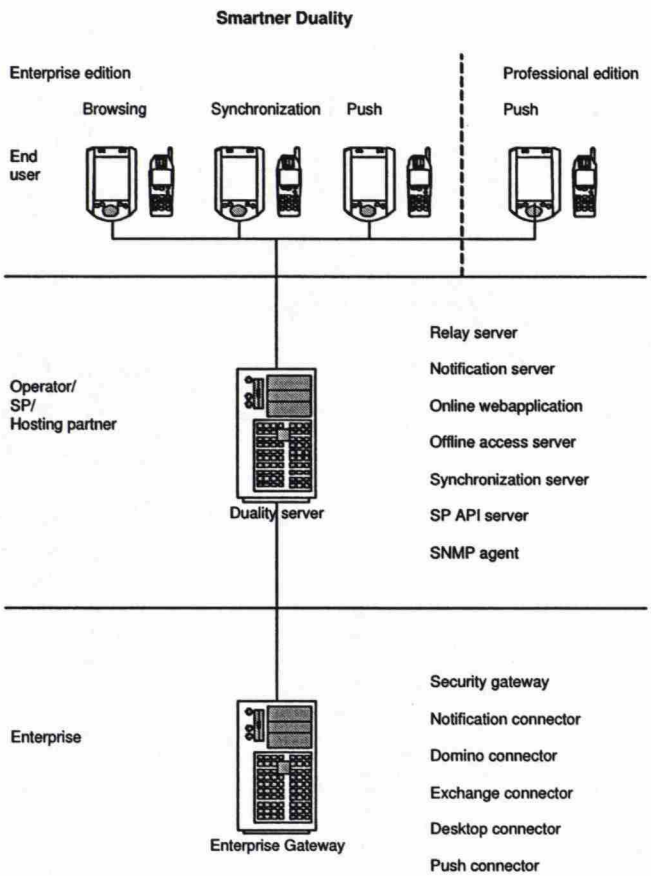

Figure 2.1: The component composition of Smartner Duality

In the case of the Professional Edition and the software installed on the device, one must keep in mind that the amount of active configurations is far greater than with the server installation, as one server can be used by thousands of end users. This must be remembered especially in the case of reconfiguration, because maintaining information about all active configurations in this layer can be very difficult or even impossible.

2.1.2 Duality Server Layer

The Duality Server is installed at the premices of an operator or a service provider. If the server is installed using a hosting partner, the actual customer can be an operator or a service provider, which has bought the solution as a hosted one. These are the actual customers of the case company and for the other layers the customers are customers of the operator, service provider or hosting partner.

The configurations on this layer are the easiest ones to keep track of, but they are also the most complex ones. Reconfiguration on this layer is easier to do than on the other layers, because there is a direct communication channel to the customer. There is a risk of interoperability problems with the end user and Enterprise Gateway layers if the reconfiguration is not done in a controlled manner.

2.1.3 Enterprise Gateway Layer

The Enterprise Gateway is installed at the premises of an enterprise customer of an operator or a service provider. The amount of these installations can be hundreds or thousands of times the amount of Duality Server installation and thus similar conditions as in the end user level Professional Edition installation apply. Reconfiguration on this layer can be very difficult to achieve and thus any reconfigurations on the Duality Server layer must be done by keeping the interoperability with this layer in mind.

This layer also introduces the most complex environment where the initial installation is done. Whereas in the end user and Duality Server layers, there is quite a good undestanding of the third party software already present at the installation site, here the environment consists of a large variety of operating systems, email servers, databases, third party libraries etc. This raises need for both clear definition of the supported environment but also a need to identify a faulty one.

2.2 Levels of Component Abstraction

This section describes the different levels of component abstraction that can be used to describe the component composition of Smartner Duality. Figure 2.1 presents the two highest levels, the highest being where the product is presented in the

three layers. The second highest level is the level where the layers are divided into subproducts, e.g. in the case of the Duality Server layer, there are Relay Server, Online Webapplication etc. The lowest level of abstraction is when the subproducts are divided into components, namely collections of compiled source code, providing a set of functionality. In future reference, this level is called the component level. The items on each level can be considered as components and they can have subcomponents either in the same level (e.g. a subproduct can have other subproducts) or on a lower level. The same configuration modeling methods should apply on any level, so the levels can be used to describe the actual implementation in reference to the products for which the methods are applied. It does not cause a need for the modeling technique to have a distinction between these levels, only to have a possibility for components to have subcomponents.

2.2.1 The Layer Level

The layer level consists of the three levels described in the previous section. This level gives the highest level of abstraction, which can be used to give a first picture of the product to outside people like potential customers. It also describes the product from an installation point of view.

2.2.2 The Subproduct Level

In the subproduct level, one layer from the previous level is divided into subproducts, which are individual parts of the product providing a function like synchronization. A subproduct can consists of other subproducts and components.

2.2.3 The Component Level

The component level represents the lowest level of abstraction in the component composition of a product. In this level, the interaction between different components is something that the configuration modeling technique must address. In the case of reconfiguration, a subset of components inside a subproduct are upgraded and the resulting subproduct component structure must be able to interact as a working configuration.

2.3 Chapter Summary

This chapter described the component composition of the case company's solution, Smartner Duality. A set of layers, which describe the composition from installation location interaction point of view, were defined. These layers are *End User Layer*, *Duality Server Layer* and *Enterprise Gateway Layer*. Three different component abstraction levels were also defined, namely *The Layer Level*, *The Subproduct Level*

and *The Component Level.* With the contents of this chapter, objective 1 (see section 1.3) can be considered reached.

Chapter 3

Scenario Requirements

This chapter lists the requirements set by the case company's product configuration scenarios (section 1.2.1). The requirements are based on knowledge acquired by interviewing representatives of different functions in the case company and by using the author's personal experience and knowledge about the product configuration scenarios. First, requirements are listed and explained for each scenario and last, all the requirements are summarized and categorized to be used as a basis for evaluating the existing modeling techniques and creating the 'ideal' model for the case company.

3.1 Reconfiguration

Reconfiguration is the most important scenario, because the other scenarios are somehow linked to it. Supportability and maintainability might lead to reconfiguration, the sales perspective leads to making an initial configuration (a special case of reconfiguration) and the future aspect acts as input for planning reconfiguration. The main requirements set by the reconfiguration scenario are evolution, backward compatibility, availability, version visibility, feature/bug fix traceability and automation.

3.1.1 Evolution

Reconfiguration can be done in two ways, by adding new components, which provide new functionality and bug fixes or by replacing old revisions of a component with new revisions. In the case company, both ways are used and thus the modeling technique must be able to support evolution in terms of being able to model new revisions of the same component.

3.1.2 Backward Compatibility

As discussed in chapter 2, there is a difference in the number of configurations existing on different layers of the product. When making reconfiguration on a layer, one must keep in mind that there might be a number of configurations on another layer which are not compatible with the current version. The best choice would be to enforce backward compatibility especially on the Duality Server layer, but it might not be possible because of additional features brought by new versions etc.

3.1.3 Availability

Because reconfiguration is usually done to an environment which is not local i.e. the actor of the scenario is usually on site at the customer premices, the product configuration model information needs to be available on site for the actor. There should not be any need to contact anyone else to get information about which components need to be upgraded, but the information should be available using the tools and components available for the actor. For an external user of the components, upgrading a configuration to a new one should not need contacting an employee of the case company.

3.1.4 Version Visibility

Version visibility means acquiring information about the versions of individual components in a configuration. In order to keep track of configurations existing in different places, one should be able to easily form a summary of any configuration installed in any place. The configuration model implementation should provide a way to summarize the version composition of a configuration in order to provide a way to determine which components need to be upgraded.

3.1.5 Feature/Bug Fix Traceability

The input for reconfiguration is always a need to gain new features or make bug fixes apply. There should be a way to determine which way the reconfiguration changes the existing configuration i.e. which are the new features available and which bugs have been fixed by upgrading to the new version. This requires there to be a way to combine features and bug fixes to a configuration and also there is a way to form a delta in terms of features and bug fixes from two configurations.

3.1.6 Automation

Reconfiguration always requires some level of manual work to be done (from someone pressing a button to activate a program doing the reconfiguration to installing the

individual components by hand). The reconfiguration task should be very straighforward and risk-free so this raises the need for high-level automation in reconfiguration. The implementation should provide means for the actor of this scenario to activate an upgrade program which examines the existing configuration and from upgrades only the needed components from the upgrade package.

3.2 Supportability and Maintainability

Supportability and maintainability scenario shares some of the requirements of the reconfiguration scenario, but the motivation for the requirements is somewhat different. As the main function of support is to keep the customer happy, it should be easy to provide the customer with the information they need. The requirements shared with the reconfiguration scenario are evolution, availability, version visibility and feature/bug fix traceability and in addition the scenario sets the requirement for multiple levels of abstraction.

3.2.1 Evolution

As bug fixes are delivered by making a new revision of a component or components, it should be possible to gain information about component revisions installed and available. Versions or revisions are also a good and understandable way of communicating issues between parties involved in support and maintenance e.g. "This bug has been reported in version 1.2.3 and it is fixed in version 1.2.5 so by installing this version you will get the bug fixed."

3.2.2 Availability

In order to be able to involve only a single person when responding to issues concerning configurations, the configuration information needs to be available to the support on-call person. The person can then respond to the support requests using this information. As means of communication, the information needs to be available to persons making decisions about maintenance of an existing configuration, because this person has not necessary been involved in making the actual changes into the maintenance release.

3.2.3 Version Visibility

As in the reconfiguration scenario, the requirement for version visibility is caused by the need to know which components need to be upgraded. In this scenario it is caused by the need to know what the customer is talking about. The actor of this scenario should be able to find out the configuration existing in the customer

premises very easily in order to know what the other comparison point is to the current situation.

3.2.4 Feature/Bug Fix Traceability

In order to give valid information about bug fixes to the customer, the actor of the scenario needs to be able to find the information about bug fixes made to the configuration installed at the customer premises. It also provides the actor a way to communicate how the bug fixes can be delivered to the customer. From a maintenance point of view, by fulfilling this requirement, there is a way to determine which open issues can be closed by making and delivering a maintenance patch to a configuration installed at a customer.

3.2.5 Multiple Levels of Abstraction

As the actor of this scenario does not have such as a high level of understanding of the components as the actor in the reconfiguration scenario, attention should be directed to the way the model provides the information about product configurations. The actor communicates directly with the customer who does not have and does not need to have an undestanding of the versions of individual components installed, but of higher level entities, like product versions. The abstraction level should follow the level which is communicated to the customer instead of the abstraction level used inside the company.

3.3 The Sales Perspective

The actor of this scenario is usually someone who is not directly linked to product development, so the abstraction level is very high when the product is concerned. The use of configuration modeling raises a possibility to use the information as sales material i.e. showing a picture of the to-be-sold solution in its environment (supported end user devices etc.). As the customer contact is usually non-technical, there should not be a need to go too deep in to detail, but to present the information in a simplified and interesting manner. As with the reconfiguration scenario, the information is needed wherever a sales event is taking place. The requirements set by this scenario are evolution, availability, multiple levels of abstraction, visualization and backward compatibility.

3.3.1 Evolution

In this scenario, evolution is mostly related to the environment and the third party software there. As new versions of specifications (e.g. WAP, SyncML), email servers (Exchange 5.5, Exchange 2000, Exchange 2003), databases (Oracle 8.1, Oracle 9)

and others become available, the modeling technique must be able to link the solution into different versions of its surrounding environment. By providing this, the sales representatives are able to respond to questions like "Does your solution support WAP1.2?".

3.3.2 Availability

The need for availability raises here mainly for the same reasons as with the reconfiguration scenario. But as the information is used for presentation, the form in which it is transported is not necessarily the same. The difference is that in this scenario, there is no existing configuration in the form of a solution but of the environment described by the customer. The implementation should allow forming a configuration "on the fly" by specifying some parts of the configuration. It should be possible to transport the information in some form to the meeting.

3.3.3 Multiple Levels of Abstraction

Like with the supportability and maintaintability scenario, the actor is communicating with a faction who does not need or even want a too deep knowledge about the components in a configuration. Also the actor himself does not necessarily need such information. The level of abstraction is even higher than with the supportability and maintainability scenario, because there is no existing solution available and the customer does not have a very deep understanding about the structure of the solution.

3.3.4 Visualization

The requirement for visualization raises from the need to present the information provided by the model to a non-technical audience. The information should be able to be presented in the form of e.g. boxes and connectors, where the level in which the to-be-sold solution can be shown in different levels of abstraction depending on the audience.

3.3.5 Backward Compatibility

Expecially in the case of after-sales one must keep in mind the existing configuration and how the possible new features affect the interoperability with the old enterprise customers of the operator or service provider. At least it should be known, which enterprise installations need to be upgraded with the upgrade of the operator or service provider installation.

3.4 The Future Aspect

Where the other scenarios usually have only two dimensions, the past and the present, this scenario adds the future. In this scenario, the actors must be able to construct information about configurations which do not exist yet. The main concern here is new features, which are added to the products. Thus feature traceability plays an important role here. As the configurations cannot be planned on individual component level, a higher level of abstraction is needed. Backward compatibility needs to be considered when planning the future, because the future solution needs to support at least some of the users supported by the earlier solution.

3.4.1 Evolution

Modeling evolution is essential to the future aspect scenario both in mapping new features to new revisions of the solution or its subproducts or components. Any indications of the solution being able to respond to the changes in its environment i.e. how it supports future versions of specifications or any third party software existing in the environment, must be able to be modeled using the technique selected.

3.4.2 Feature/Bug Fix Traceability

Actually for the future aspect, bug fixes do not play such an important role as the new features. But bug fixes need to be considered in the form of bug fixes expected of third party components used. For the new features, the way in which they can be added to existing configurations need to be considered as soon as it is feasible. This scenario is closely linked with the sales perspective, because the sales people need to be able to sell features which do not exist yet, but are already planned.

3.4.3 Multiple Levels of Abstraction

It is not necessarily possible to look at the new features in the level of new components and changes into existing components. Thus the product configuration model implementation needs to offer a way to create model elements in the form of new features or some kind of representation of a new feature.

3.4.4 Backward Compatibility

This has an important role in planning the future. As it is difficult to know all the possible configurations in the field of end user and Enterprise Gateway layers, there should be some level of planning related to the possible interoperability problems with the planned new features and the existing end user and Enterprise Gateway layer configurations.

3.5 Summary of Requirements

The following lists the requirements set by the scenarios. In order to make decisions about which requirements need more attention than others, the requirements are prioritized. The prioritization is done using the following criteria:

- For each scenario where the requirement is present, it receives +1 priority.
- As the reconfiguration scenario is considered the most important one, the requirement receives extra +1 priority.

The requirements, their occurrence in the scenarios and their priority calculated using the criteria above is presented in table 3.1.

Requirement	**Set by Scenarios**	**Priority**
Evolution	Reconfiguration Supportability and Maintainability The Sales Perspective The Future Aspect	5
Feature/Bug Fix Traceability	Reconfiguration Supportability and Maintainability The Future Aspect	4
Backward Compatibility	Reconfiguration The Sales Perspective The Future Aspect	4
Availability	Reconfiguration Supportability and Maintainability The Sales Perspective	4
Version Visibility	Reconfiguration Supportability and Maintainability	3
Multiple Levels of Abstraction	Supportability and Maintainability The Sales Perspective The Future Aspect	3
Automation	Reconfiguration	2
Visualization	The Sales Perspective	1

Table 3.1: The Requirements Set by the Scenarios

It should be noted that the selection of giving an extra +1 to the requirements set by the reconfiguration scenario gives most of them a very high priority. But as mentioned in section 3.1, the other scenarios are somehow linked to the reconfiguration scenario, so there is reason to give these requirements high priority.

3.6 Chapter Summary

This chapter listed the requirements that were derived from the case company's product configuration scenarios (see section 1.2.1). The requirements were also prioritized. The requirements are in priority order: *Evolution*, *Feature/Bug Fix Traceability*, *Backward Compatibility*, *Availability*, *Version Visibility*, *Multiple Levels of Abstraction*, *Automation* and *Visualization*. These requirements will be used to evaluate existing product configuration modeling techniques and the implementation created by the study. With this information, objective 2 for the study (see section 1.3) can be considered reached.

Chapter 4

Existing Configuration Modeling Techniques

This chapter presents existing configuration modeling techniques and evaluates them against the requirements presented in chapter 3. It also discusses the currently used methods in the case company and their applicability in responding to the requirements. First, the modeling techniques are described and their applicability is evaluated and last, the model evaluation is summarized as input for the description of the 'ideal' modeling technique.

The evaluation of a modeling technique is conducted by first giving an overall evaluation of the information available about the modeling techniques and then evaluating it against each requirement individually. The evaluation is completed by giving a summary of the evaluation against the requirements. In this summary and later in the summary of all techniques, the way how a requirement is fulfilled is rated using the following criteria:

- **Supported** The requirement is fulfilled by the modeling technique.
- **Partly Supported** The modeling technique possesses some qualities that promote somehow the fulfilment of the requirement but the requirement is not completely fulfilled.
- **Unsupported** It can be stated that using the modeling technique, this requirement cannot be fulfilled.
- **Undefined** The modeling technique does not specify anything that is related to the requirement, but does not indicate any reason why it could not fulfil the requirement.

The first three modeling techniques, conceptual model of components and their relationships, rule-based component selection and component selection based on existing configurations are based on a paradigm categorization used in a survey of

product configuration frameworks (Daniel Sabin and Rainer Weigel, 1998). The corresponding paradigm names in the survey are model-based reasoning, rule-based reasoning and case-based reasoning. The fourth modeling technique, no reconfiguration, is not actually a modeling technique itself, but an idealism to always conduct installations from scratch. It is a very common practice in the industry (Tomi Männistö, Timo Soininen, Juha Tiihonen and Reijo Sulonen, 1999). It is included in the evaluation to gain an understanding whether a complex configuration modeling is even needed or can a simple way of thinking be the correct way to respond to the requirements. The fifth modeling technique, reconfiguration through patches, represents the way product configuration is handled in parts of the case company's solution. It is included in the evaluation in order to find out whether a new modeling technique is needed or can the existing way be extended to cover the whole solution.

4.1 Conceptual Model of Components and Their Relationships

This section describes and evaluates a modeling technique studied in the Sarcous project (http://www.soberit.hut.fi/sarcous/english/index.html) of the Laboratory of Software Business and Engineering at Helsinki University of Technology. The technique is also called as model based configuration (Gunilla Sivard, 2000) and model-based reasoning (Daniel Sabin and Rainer Weigel, 1998).

4.1.1 Modeling Technique Description

The technique is based on a conceptualization of configuration knowledge (Juha Tiihonen, Timo Lehtonen, Timo Soininen, Antti Pulkkinen, Reijo Sulonen and Asko Riitahuhta, 1999). The conceptualization is based on concepts used in Architecture Description Languages (ADLs) (Timo Asikainen, 2002). The technique defines *a configuration task*, where *a configuration* is formed by combining *a configuration model* and customer requirements using *a configuration engine*. This operation is depicted in figure 4.1.

A configuration model defines the set of legal product individuals by explicitly defining the components out of which the individuals can be constructed and the relationships between the components. It also defines a set of *constraints* i.e. rules that must be followed when combining the components. The customer requirements are represented in the form of constraints required by the customer and the configuration engine checks whether a valid configuration can be constructed. A valid configuration here is one that implements the customer requirements without breaking any of the constraints defined by the configuration model.

The basic technique does not offer very good support for evolution so it was extended (Tero Kojo, Tomi Männistö and Timo Soininen, 2003) using a subset of de-facto standard ontology of product configuration knowledge (Timo Soininen,

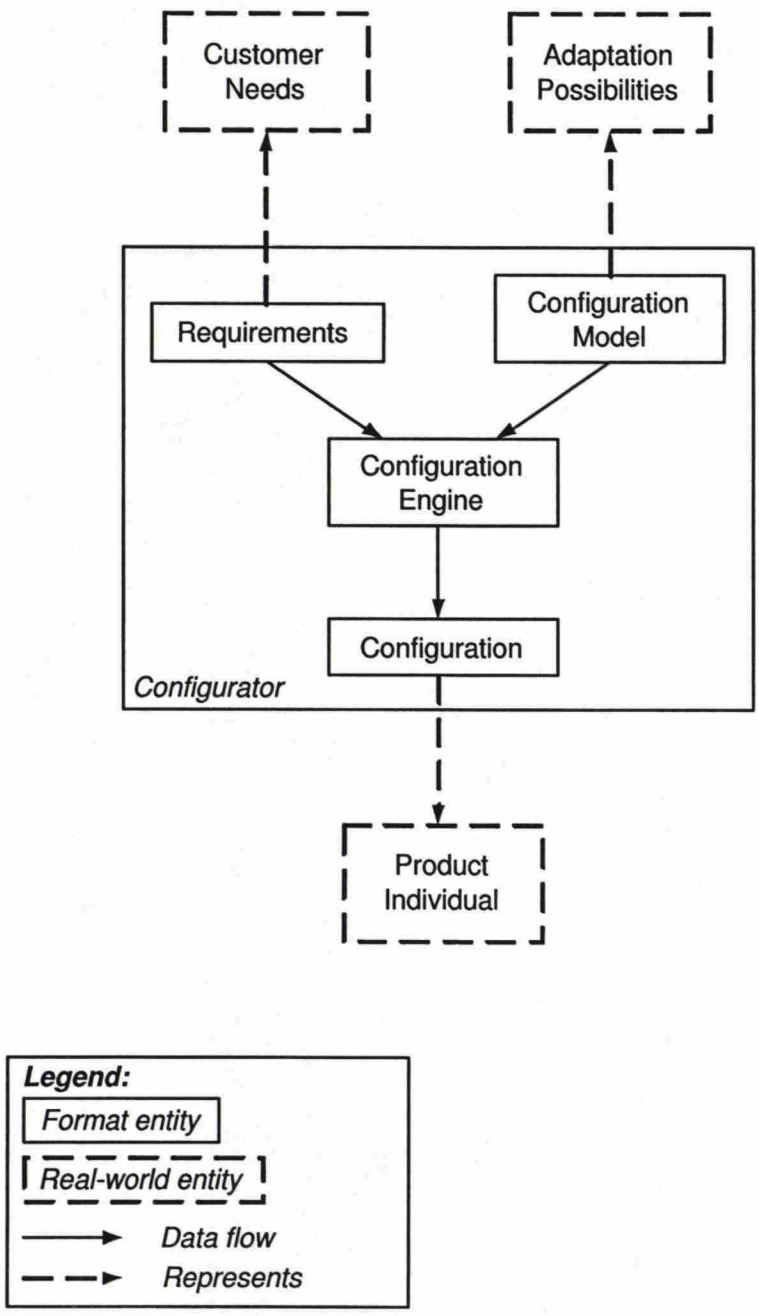

Figure 4.1: A Configuration Task (Timo Soininen, 2000)

Juha Tiihonen, Tomi Männistö and Reijo Sulonen, 1998). The conceptualization of the extended modeling technique can be seen in figure 4.2. The notation used in the figure can be seen from for example (Fowler and Scott, 2000).

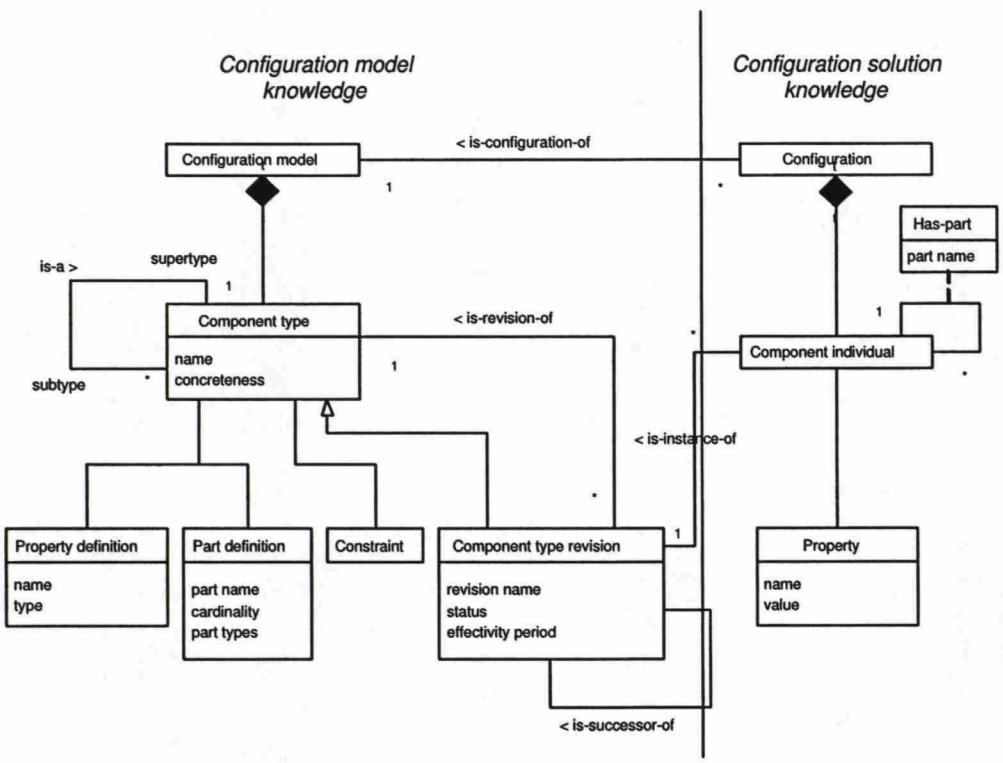

Figure 4.2: The Conceptialization of the Extended Modeling Technique (Tero Kojo, Tomi Männistö and Timo Soininen, 2003)

The left hand side of figure 4.2, including configuration model, component type, property definition, part definition, constraint and component type revision, represents *configuration model knowledge* (Timo Soininen, Juha Tiihonen, Tomi Männistö and Reijo Sulonen, 1998). This specifies how a configuration can be formed i.e. the entities it can have and the rules how the entities can be combined. The right hand side, including configuration, component individual and property, represents *configuration solution knowledge* (Timo Soininen, Juha Tiihonen, Tomi Männistö and Reijo Sulonen, 1998). This specifies a real-life configuration which is formed using the information provided by the configuration model knowledge. For example in case of a car and its engine, a configuration model description would be that a car must have an engine and the engine can be one of the following: 1.0 litre, 1.2 litre and 1.4 litre version. A configuration solution knowledge representation of a car would be a car constructed following the component structure and rules presented in the configuration model knowledge, e.g. a car equipped with a 1.2 litre

engine.

When using this modeling technique, there must first be a mapping of the information available from the actual components and their relationships in a solution to the conceptualization in figure 4.2. One must have some means of giving the requirements as input to a configurator which then forms a product or solution individual based on the requirements and the conceptualization of the solution being modeled.

In a prototype implementation of the modeling technique (Tero Kojo, Tomi Männistö and Timo Soininen, 2003) for Linux Familiar (http://familiar.handhelds.org/) used a Product Configuration Modeling Language (PCML) (Juha Tiihonen, Timo Soininen, Ilkka Niemelä and Reijo Sulonen, 2002) to represent the conceptualization of the components of Linux Familiar and their relationships. It used a WeCoTin [1] configurator, which has a user interface to gather the user requirements (selection of components to be installed) to acts input for creating a configuration. Other prototype implementations of tools for model-based product configuration exist also, like the ConBaCon [2] system, which is constraint-based approach to model-based product configuration developed by U. John and U. Geske (U. John and U. Geske, 1999b,a) and KONWERK (Andreas Gnter and Lothar Holz, 1999). The applicability of these tools must be studied if this modeling technique is selected as the 'ideal' modeling technique.

4.1.2 Evaluation

The technique of having conceptual model of components and their relationships is evaluated first on a general level based on the information available about the technique and the possible tools available to be used to implement the technique. The evaluation is summarized to form a basis of a summary of all techniques.

General Evaluation

The Sarcous project (http://www.soberit.hut.fi/sarcous/english/index.html) has produced a lot of material about conceptual modeling. The modeling technique has not yet been applied in many cases (Juha Tiihonen, Timo Lehtonen, Timo Soininen, Antti Pulkkinen, Reijo Sulonen and Asko Riitahuhta, 1999; Tero Kojo, Tomi Männistö and Timo Soininen, 2003) and only some of them in the software industry (Tero Kojo, Tomi Männistö and Timo Soininen, 2003). This raises concern whether the modeling technique can be implemented in a rapidly evolving mobile software industry. There are also tools available to help in implementing the modeling technique like the PCML for describing the conceptualization of a solution and the WeCoTin configurator. These tools lack real life reference cases from software

[1] Stands for Web Configuration Technology
[2] Constraint Based Configuration

industry (more references in traditional industry) so the applicability of these tools to the task at hand must be studied.

Evaluation Against Scenario Requirements

Evolution

The extended version of the conceptualization in figure 4.2 provides some kind of support for evolution but as stated in (Tero Kojo, Tomi Männistö and Timo Soininen, 2003), the applicability of these extensions is not elaborated. Also PCML and WeCoTin tools do not provide direct support for evolution.

Feature/Bug Fix Traceability

The modeling technique offers the *is-successor-of* relation between different component type revisions. If there is a way to link features and bug fixes to a certain component type revision (see figure 4.2) then by following the successor relation between different component type revisions, this requirement can be satisfied.

Backward Compatibility

By using the effectivity periods and constraints, backward compatibility can be achieved at least at the level of knowing when it will be lost. It must be stated here that one cannot assume to have backward compatibility for an indefinite period of time but for a defined time period. By keeping track of the effectivity periods of different revisions one can have a view to the backward compatibility of the solution.

Availability

Availability mainly causes a requirement to have compact and easy-to-use tools available to use the information provided by the modeling technique. As discussed earlier, there are some tools available, but their applicability to the situations where the information is needed to be available (see chapter 3) needs to be studied.

Version Visibility

In the modeling technique, each component individual has an *is-instance-of* relation with a component type revision. This results in having version information available for each component. However, the requirement description raises a need to have version information on all components of a certain configuration. The tools existing for this technique offer a chance to create all possible configurations and on the basis of user selection, make one configuration, but gathering information on an existing configuration is not defined. Creating such a tool should not be a complex task, because the version information of individual components is available.

Multiple Levels of Abstraction

The conceptualization in figure 4.2 allows component individuals to have other component individuals as parts. Basically this ability offers a chance to describe the components of a solution in different abstraction levels using the part relationships. Using the levels of abstraction of the case company's solution in chapter 2, a subproduct can have its components as part, while the subproduct is also a component individual. Also research done in (Tomi Männistö, Hannu Peltonen, Timo Soininen and Reijo Sulonen, 1998) supports using multiple levels in this modeling technique.

Automation

As well as availability, automation is also related to the tools available for the technique. In order for the technique to be automated, there needs to be a tool to create the conceptual model automatically out of the information available from the case company's solution and a configurator, which can select a valid configuration according to some selection criteria (like user requirements). WeCoTin seems to have good qualities in creating configurations from PCML descriptions, but transforming the information available from the case company's solution to PCML might be quite laborious.

Visualization

The WeCoTin user interface described in (Tero Kojo, Tomi Männistö and Timo Soininen, 2003) offers some level of visualization. However, in order to use the information for e.g. sales purposes, it must be in more understandable format i.e. it should provide information understandable to all possible user groups like sales representatives and potential customers. One way of giving such information is through graphs. In order to use graphs, a graph-based user interface needs to be implemented, using the information available provided by the conceptualization.

Summary

The modeling technique has quite good support for all the high priority requirements. But because of the lack of knowledge about the existing tools' applicability to be used for availability, automation and visualization, it could cause a significant amount of implementation work to be done before the modeling technique can be taken into use. The summary of the fulfilment of the requirements is presented in table 4.1.

4.2 Rule-based Component Selection

This section describes a technique where component selection is based on rules. The technique is also referred to as rule based configuration (Gunilla Sivard, 2000) and

Requirement	Fulfilment of the Requirement
Evolution	Partly Supported
Feature/Bug Fix Traceability	Partly Supported
Backward Compatibility	Supported
Availability	Partly Supported
Version Visibility	Supported
Multiple Levels of Abstraction	Supported
Automation	Partly Supported
Visualization	Partly Supported

Table 4.1: Requirement Fulfilment in Conceptual Model of Components and Their Relationships

rule-based reasoning (Daniel Sabin and Rainer Weigel, 1998).

4.2.1 Modeling Technique Description

The technique is based on having a generic structure of the components and then a set of *rules* defining the alternatives to the generic structure. A rule represents both directed relationships (i.e. domain knowledge like component relationships and compatibility) and actions (selections based on user requirements etc.) (Daniel Sabin and Rainer Weigel, 1998). The rules are presented in the following fashion:

if *condition* **then** *consequence*

Thus executing a rule always changes the state of the configuration, affecting the execution of a number of other rules (by affecting their conditions).

In addition to the compatibility rules, there is also some kind of conceptual model about the basic component structure. This structure may be quite similar as the left hand part of the conceptual model in figure 4.2. The rules are built on top of this.

To clarify the idea behind rule-based configuration, a simple example from traditional industry is presented. Assume a car being constructed of parts. The base model consists of the components that a car is constructed from. Not to go too deep into detail, a subset of traditional car components are presented here. Let's say a car is composed of the following components:

- Engine, a selection of two different engines, a basic 1.6 litre and a sporty 2.0 litre version.
- Wheels, basic wheels and low-profile racing wheels.
- Radio, with or without a CD-player.
- Gas tank, 40 litre or 60 litre.

The domain knowledge consists of the following issues:

- A car always has an engine, wheels, a radio and a gas tank
- If a 2.0 litre engine is selected, a 60 litre gas tank is needed.

The customer can select from the following user requirements for a car:

- luxury, radio with CD-player
- sporty, 2.0 litre engine and low-profile racing wheels.

Figure 4.3 presents a configuration creation from the base model and the rules presented above.

The user requirements of luxury and sporty affected the selection of engine, wheels and radio. The selection of the gas tank was affected by the selection of the engine. Here one can notice that the rules also have relationships with each other as the gas tank cannot be selected before the engine is selected.

In (Tommi Syrjänen, 1999), Tommi Syrjänen used a rule-based configuration modeling language based on a declarative rule language (Timo Soininen and Ilkka Niemelä, 1999) to create a presentation of the package descriptions of Debian Linux. He used existing software called *smodels* (Ilkka Niemelä and Patrik Simons, 1995) and *lparse* (Tommi Syrjänen, 1998) to model the package descriptions. The applicability of these tools can be researched if this technique is selected.

4.2.2 Evaluation

The technique of using a generic structure of components and rules to create configurations is evaluated first on a general level based on the information available about the technique and the possible tools available to be used to implement the technique. The evaluation is summarized to form a basis of a summary of all techniques.

General Evaluation

This modeling technique more or less defines all available solution options i.e. the possible solutions are those that are formed from the basic structure following the rules. In a complex solution there is a large number of components and rules. The rules might have complex relations with one another. When the structure changes (e.g. adding new functionality by adding new components) then the whole rule structure needs to be evaluated to determine which rules need to be changed because of the modification (Gunilla Sivard, 2000). This makes maintaining the modeling technique very laborious.

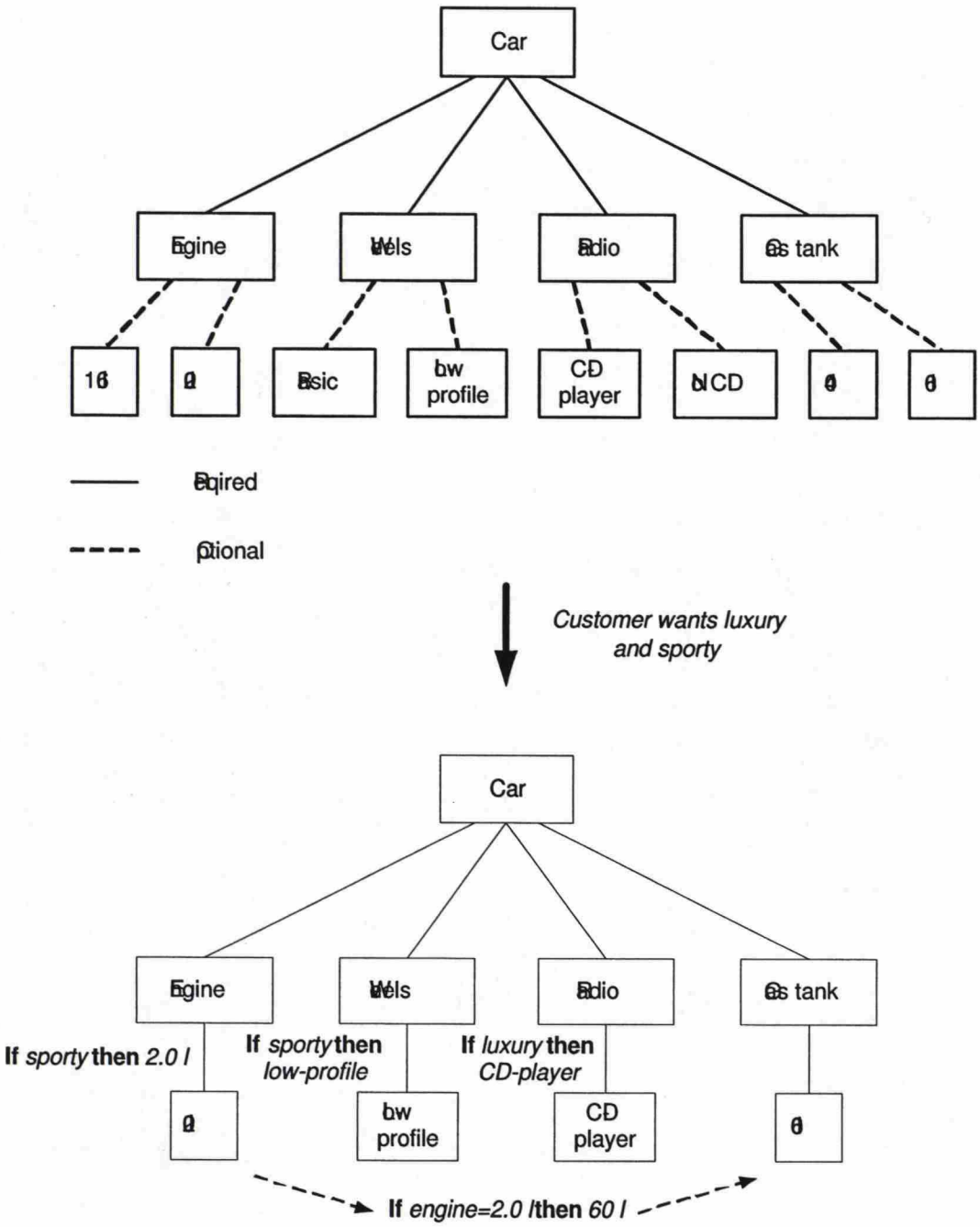

Figure 4.3: Example of Rule-based Product Configuration

Another problem raises from the fact that there is no separation in the way in which domain knowledge and customer requirements are presented. This leads to the fact that when updating the structure in case of e.g. additional customer selection (such as new pricing criteria), the rules defining domain knowledge possibly need to be updated also and vice versa.

Evaluation Against Scenario Requirements

Evolution

Because of the maintenance problem described in the general evaluation, modeling evolution can be very tricky using this technique. Adding a new revision of a component basically creates one more selection of a single component and the selection can also cause a need for other components to be upgraded. Depending on the complexity of the existing rule structure this can cause an uncontrolled amount of new rules and changes to existing rules.

Feature/Bug Fix Traceability

As the technique lacks presentation of a functional property (Gunilla Sivard, 2000), combining features and bug fixes to certain configurations and tracing them needs to be done using selection rules. As a selection rule is always related to a number of actions made based on other customer selections and the rules coming from the domain knowledge, it may require a lot of work before a certain configuration containing a certain feature or bug fix can be recreated.

Backward Compatibility

As backward compatibility resembles a rule in practice, it can be modeled using a rule. Adding rules for backward compatibility adds to the general complexity of the model affecting many other rules.

Availability

There is little knowledge about tools using this modeling technique. Some expert systems exist, but the applicability of these tools to the problem area represented in the scenarios is unknown. Thus implementation must be done from scratch and as the technique does not offer all the other requirements, implementing such a tool might be a waste of effort.

Version Visibility

The concept of version is missing from the technique and as it does not fit into the concepts defined in the technique, the requirement cannot be fulfilled.

Multiple Levels of Abstraction

This requirement can be considered supported, because one can move from one abstraction level to another using rules (if you select a subproduct, you automatically select its components).

Automation

If the generic model and the rules are available, then forming a configuration is quite straightforward. This would only require some means of calculating the configuration from the rules provided. But as creating automation is very much dependent on the tools available, responding to this requirement depends on the applicability of the tools.

Visualization

The technique offers a chance to create a user interface where there is a possibility to fill in the customer requirements and create a configuration based on the domain knowledge rules and inputted customer requirements.

Summary

The modeling technique provides very little support for the high priority requirements. Only some of the low priority requirements can be considered partly supported. A summary of requirement fulfilment is presented in table 4.2.

Requirement	Fulfilment of the Requirement
Evolution	Unsupported
Feature/Bug Fix Traceability	Undefined
Backward Compatibility	Partly Supported
Availability	Unsupported
Version Visibility	Undefined
Multiple Levels of Abstraction	Supported
Automation	Partly Supported
Visualization	Partly Supported

Table 4.2: Requirement Fulfilment in the Rule-based Component Selection

4.3 Component Selection Based on Existing Configurations

This section describes a technique where component selection in configurations is based on experience with existing configurations. The technique is based on a more general Case-based Reasoning (CBR) paradigm (Roger C Schank and Riesbeck, 1994).

4.3.1 Modeling Technique Description

In this technique, creating new configurations is based on existing configurations existing at customers, test installations etc. The problems in creating the new

configurations are solved based on similar problems that have been solved with the existing configurations. The technique relies on the assumption that similar problems have similar solutions. The reasoning process is depicted in figure 4.4.

In figure 4.4 there has been component interoperabilities between components in an existing configuration A. The way these problems have been solved has been stored in a knowledge base for further use. A new configuration B is formed with similar component connections and problems as in configuration A. These problems are solved using the knowledge on the problem solving in configuration A. The configuration creation process is done in four phases (Daniel Sabin and Rainer Weigel, 1998):

- *Input customer requirements* Find out the customer requirements to find a similar existing configuration.
- *Retrieve a configuration* Search the knowledge base for a configuration matching the customer requirements.
- *Adapt the case to the new situation* Use the information on the existing configuration to create the required configuration.
- *Store the new configuration* Each time a new configuration is created it is stored into the knowledge base to be used with new configurations.

4.3.2 Evaluation

The technique of using existing configurations as a basis for forming new configurations is evaluated first on a general level based on the information available about the technique and the possible tools available to be used to implement the technique. The evaluation is summarized to form a basis of a summary of all techniques.

General Evaluation

The technique relies heavily on the assumption that similar problems have similar solutions. This causes mainly three challenges:

- How to verify that the problems are similar?
- How to adapt the existing solution to create an exact solution to the problem at hand?
- How to verify that the created solution actually solves the problem at hand?

In order to overcome these challenges, one must first create some method of describing existing configurations and customer requirements in a way that they can be compared and determine whether there is a way to say that two cases are equal

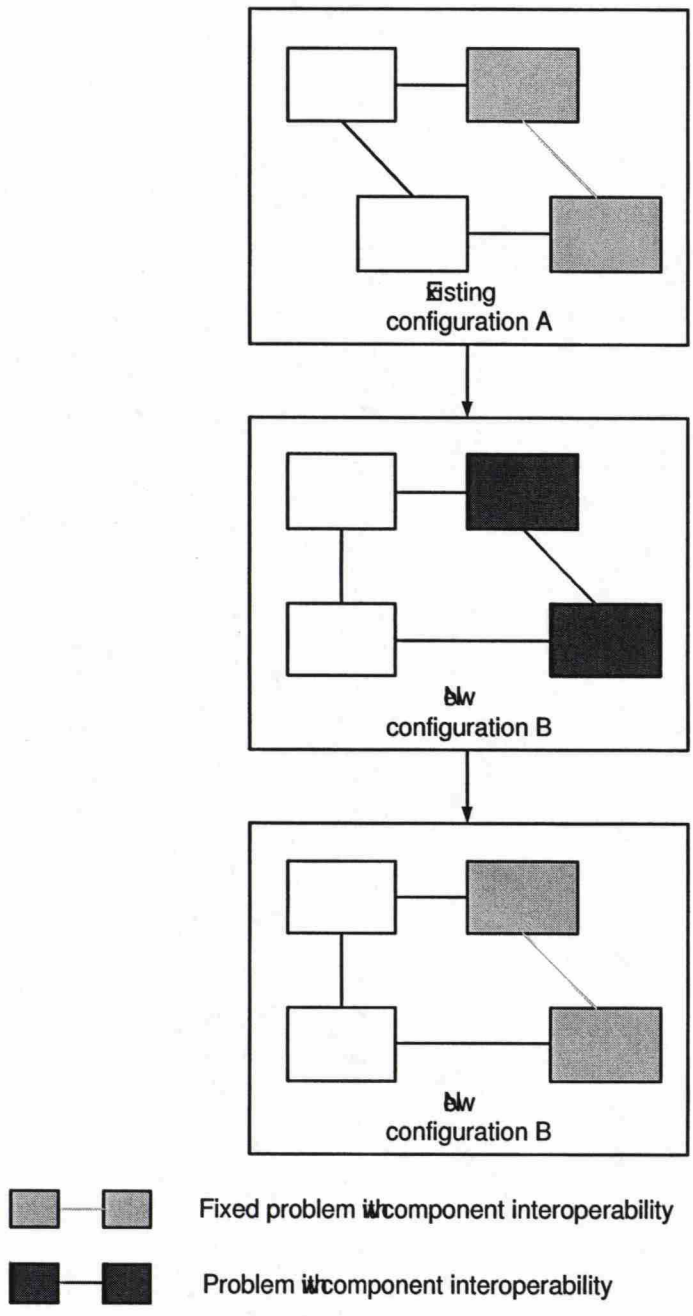

Figure 4.4: The Case-based Reasoning Process

in all aspects. The configuration structure must be flexible so that an existing configuration can be adapted to support the requirements of a new configuration. There must also be a way to find out how the existing configuration needs to be changed. The created configuration must also somehow be evaluated against the original requirements.

Maintaining a knowledge base can cause a problem because in an evolving environment where existing configurations are upgraded quite often there is a risk of having outdated information in the knowledge base. Basically maintaining such a knowledge base would require some level of a policy on how and when the base is updated.

Evaluation Against Scenario Requirements

Evolution

The technique is based on creating new configurations based on old ones, so it does not offer very good support for evolution. But if new features are used as user requirements, existing configurations can be evolved with new features by adding them to the requirements.

Feature/Bug Fix Traceability

As all existing configurations exist in the knowledge base, one can easily trace back to the existing configuration and its features and bug fixes if they are properly documented in the knowledge base.

Backward Compatibility

This modeling technique offers a chance to take backward compatibility into account, because new configurations are based on old ones. The adaptation rules that are used to create new configurations, must be able to take into account parts of the solution where backward compatibility is needed.

Availability

The tool support for this technology is not very good. The knowledge base would probably contain large amounts of data, and having it available in all the places described in the scenarios would require some kind of client-server solution. However, ideologically the technique supports such an approach.

Version Visibility

Because all existing configurations and the newly created ones are documented, version visibility can be achieved. This requires, however, that the versions of the components are also documented, not just the component composition.

Multiple Levels of Abstraction

This depends on the way the information about existing configurations is stored. The technique itself does not contain any indication as to how the configuration information should be stored.

Automation

Automation is possible using the technique although there is no tool available. Finding an existing configuration from the knowledge base, adapting it to respond to the requirements and verification can be done automatically if all the steps have clearly defined semantics.

Visualization

Visualizing the information is largely based on the way in which the information is stored. The level of visualization is largely based on how the information can be displayed.

Summary

The modeling technique provides quite good support for the high priority requirements. Fulfilment of many requirements is based on the way the data is stored into the knowledge base. A summary of requirement fulfilment is presented in table 4.3.

Requirement	Fulfilment of the Requirement
Evolution	Partly Supported
Feature/Bug Fix Traceability	Supported
Backward Compatibility	Supported
Availability	Partly Supported
Version Visibility	Undefined
Multiple Levels of Abstraction	Undefined
Automation	Partly Supported
Visualization	Undefined

Table 4.3: Requirement Fulfilment in Component Selection Based on Existing Configurations

4.4 No Reconfiguration

This section describes a technique where the configuration is always done "from scratch" and no reconfiguration is conducted.

4.4.1 Modeling Technique Description

The technique can be based on any of the other techniques described in the earlier sections. The only requirement is to have some kind of a way to create a configuration according to customer requirements. In this technique, there never is an existing solution anywhere from the point of view of the modeling technique. There

might be some kind of a way to store the original customer requirements if a solution has already been delivered.

4.4.2 Evaluation

The technique of no reconfiguration is evaluated first on a general level based on the information available about the technique and the possible tools available to be used to implement the technique. The evaluation is summarized to form a basis of a summary of all techniques.

General Evaluation

The technique does not take into account any installation specific data created during the usage of the solution (e.g. database data, customer specific customizations, configuration data) and their compatibility with the new configuration.

If the configuration is done to a solution individual which would normally be subjective to reconfiguration, the requirements of the original solution have to be stored somewhere in order to be sure that the new configuration also satisfies these requirements (if not agreed that some requirements are not valid anymore). This raises a question of how detailed can a customer requirement be in order for a new configuration to be able to satisfy it the same way from the viewpoint of the customer.

Evaluation Against Scenario Requirements

Evolution

As the new configuration is in no way related to the old one (except for the customer requirements), there is no link between different revisions of components.

Feature/Bug Fix Traceability

This is dependent on the way features and bug fixes are documented in the user requirements. A general problem here arises with bug fixes, because usually in the case of a severe bug found in a solution delivered to a customer, it needs to be fixed quickly. Delivering a new solution in a case of delivering a bug fix would be somewhat an overkill way of handling the situation. But if all the bug fixes and features are documented in the customer requirements and a new configuration is created according to these requirements, then this requirement is fulfilled.

Backward Compatibility

As the technique is based on always delivering a new configuration, then this requirement cannot be fulfilled. The nature of the case company's solution

described in chapter 2 shows that there are parts of the solution which cannot be upgraded so easily.

Availability

In terms of availability the technique is based on keeping up with customer requirements, which is a widely researched area in the industry. By defining what is needed from a customer requirement in order to use it for product configuration, this requirement can be fulfilled.

Version Visibility

This can be achieved by always documenting the versions in a newly created configuration or by allowing a configuration to tell the component versions it contains.

Multiple Levels of Abstraction

This technique does not specify how a configuration is described. One representation of the configuration is available which is the requirements used to construct it.

Automation

This is not defined by the technique itself, because the way the requirements are used to create a configuration is not defined.

Visualization

As the use of this technique requires that the customer requirements used to create the configurations are stored somewhere, they can be used to describe an existing configuration. However, this limits the representation to a list of features.

Summary

The modeling technique provides quite poor support for the high priority requirements. As the way the configurations are formed from the requirements is not defined, lot of the requirement fulfilment is dependent on the way it is handled. A summary of requirement fulfilment is presented in table 4.4.

4.5 Reconfiguration through Patches

This chapter presents a modeling technique, which is used in the case company for parts of the solution.

Requirement	Fulfilment of the Requirement
Evolution	Unsupported
Feature/Bug Fix Traceability	Partly Supported
Backward Compatibility	Unsupported
Availability	Partly Supported
Version Visibility	Supported
Multiple Levels of Abstraction	Unsupported
Automation	Undefined
Visualization	Partly Supported

Table 4.4: Requirement Fulfilment in No Reconfiguration

4.5.1 Modeling Technique Description

In this technique, the initial configuration is made from a released product individual which fulfils the customer requirements. It is usually made with the latest release available and during the sales process, the customer is convinced that this is what they need. Reconfiguration, which is in this case mostly related to delivering bug fixes, is done by delivering patches to the initial configuration. A patch version is a version, which basically contains the same functionality than the original version, but with some parts upgraded.

In the patch versions, compatibility with the initial configuration is achieved by making the changes into those versions of the components in the solution which were part of the original configuration. This is done by making a *branch* (Per Cederqvist et al., 2003) into the version control system (like the Concurrent Versions System (CVS) (Per Cederqvist et al., 2003)), which then starts to live its own development cycle. Later the changes made into a branch can be *merged* (Per Cederqvist et al., 2003) into the main trunk version in the version control system. Branching and merging is depicted in figure 4.5. Branching can be done in many levels, but as in a version control system, individual files are considered, the level of branching must be decided by the person making the branch (whether to branch a single file or all files contained by a component).

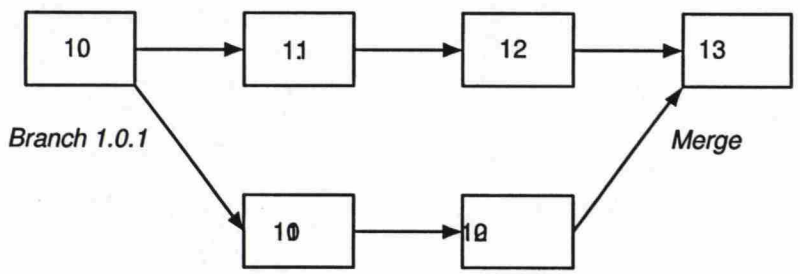

Figure 4.5: Branching Versions

In figure 4.5, version 1.0 is the initial version of a component. As part of normal component evolution (new features, refactoring, new relationships to other components etc.) versions 1.1 and 1.2 are created of the component. A bug is found from version 1.0 at a customer where this version was installed. Version 1.2, which is the current development version, contains a lot of new functionality and it requires lots of effort to create a patch which contains version 1.2 and all other changes required by the possible new component relationships. Thus, a branch 1.0.1 is created from version 1.0 and the bug fix is made there. By upgrading the customer configuration to contain the branch version 1.0.1, there is no need to make any additional upgrades or investigations about possible incompatibility issues. As part of further development, there might be a branch version 1.0.2 out with a more sophisticated bug fix to the original bug in version 1.0. At some point there might be a need to incorporate the bug fix to the main development version of the component. This is done by merging the changes made in the branch to the current development version. Merging requires checking that the changes made in the branch do not affect the changes made in the main trunk, and if they do, resolve these issues. This usually goes into the source code level. There are tools available for merging, but if the same code sections are changed in the main trunk and in a branch, the merging can be quite difficult. Maintaining information about code level changes may prove to be extremely laborious. In practice, making merging would require consulting people that have made the changes.

4.5.2 Evaluation

The technique of reconfiguration through patches is evaluated first on a general level based on the information available about the technique and the possible tools available to be used to implement the technique. The evaluation is summarized to form a basis of a summary of all techniques.

General Evaluation

This technique is very much based on delivering bug fixes to existing configurations. Its applicability to reconfiguration in general can be questioned, as then any modifications must be done so that they can be upgraded to any configuration in the field. This would require some means of keeping record of all configurations delivered and making branches for each of these configurations per each modification (measured in features, bug fixes etc.). Basically this would require maintaining (amount of active configurations) * (amount of measurable modifications) branches. As the definition of a measurable modification is very vague, the amount of branches is also hard to determine. If product variation is high, the amount of active configurations is also high.

Using this technique with a solution with high product variability and very low level definition of a measurable modification (causing a large amount of measurable

modifications) would cause the network of branches to quickly become uncontrollable. This would also make merging more difficult, because changes come from many different branches and can even be ambiguous with each other. Determining when to merge the branches to the main trunk is not an easy task, basically it would be a part of the product development process in the company.

CVS is a very powerful tool in handling the branching and merging in the individual source file level. However, it does not offer any means for automating the process of creating patch versions, which has to be controlled by the company personnel involved in product development. Comparing branch version modifications to the main trunk version must also be done by hand. Tools exist to find the differences in the files in terms of source file lines, but the idea behind the changes must be known by the person making the merge.

Evaluation Against Scenario Requirements

Evolution

The technique does not work very well with evolution. This is because all development must be done based on all possible existing variations of the solution. The resulting network of branches may grow uncontrollable if product variation is high and there are a lot of measurable modifications (large fixes, new features etc.).

Feature/Bug Fix Traceability

The requirement is fulfilled in this modeling technique. As all new features and bug fixes are made by branching, following the branch structure allows tracing bug fixes and features. This requires good version visibility, because one needs to know which patches have been applied to each configuration. CVS offers a chance to create branches with a name, so if a valid naming convention (describing what the branch provides) is used, one can use the branch names to trace features and bug fixes.

Backward Compatibility

If the patches are made in a way which does not affect component relationships, this requirement is achieved. However, as all bug fixes cannot necessarily be made this way (such as bugs related to communication between different product layers), there might be a need to upgrade parts of the product where backward compatibility is needed. The technique allows this to be done in a controlled fashion.

Availability

The requirement is not defined by the technique itself. It is very dependent on the supporting tools available in the company (such as bug reporting systems). If the bug fixes and their availability in patches is documented in a system

which is available to anyone needing the information (support personnel, customers etc.) this requirement can be fulfilled.

Version Visibility

The requirement is not defined by the technique itself, but it is needed in order to fulfil other requirements (such as feature/bug fix traceability). In order to know which configuration exists at a customer, one must be able to find out the base version and delivered patches.

Multiple Levels of Abstraction

Branching can be done in many levels (from individual source files to solutions), depending on how the source files are divided into components, subproducts, solutions etc.

Automation

The tools available help to create some level of automation, but the final steps must be conducted by hand (such as merging). Branching may also be automated (e.g. a branch is created automatically in case of a bug report being filed).

Visualization

Some CVS clients offer a chance to create visualizations out of the information stored in CVS itself, but the information is usually based on information based on a single file. Creating visualizations from data in different levels of abstraction requires a tool of its own.

Summary

The modeling technique offers some support for the high priority requirements. However, as evolution is unsupported, the technique might end up being unusable for most of the problem areas in the scenarios. Support for many requirements also remain undefined, which may result in a great amount of extra implementation work. A summary of requirement fulfilment is presented in table 4.5.

4.6 Summary of Modeling Technique Evaluation

This section presents a summary of all the modeling technique evaluations and some criteria for selecting the 'ideal' modeling technique for chapter 5.

4.6.1 Requirement Fulfilment

The modeling technique evaluation in terms of requirement fulfilment is summarized in table 4.6.

Requirement	Fulfilment of the Requirement
Evolution	Unsupported
Feature/Bug Fix Traceability	Supported
Backward Compatibility	Partly Supported
Availability	Undefined
Version Visibility	Undefined
Multiple Levels of Abstraction	Partly Supported
Automation	Partly Supported
Visualization	Unsupported

Table 4.5: Requirement Fulfilment in Reconfiguration Through Patches

Requirement	Conc. Model	Rule-based	CBR	No Rec.	Patches
Evolution	+	-	+	-	-
Feature/Bug Fix Traceability	+	+-	++	+	++
Backward Compatibility	++	+	++	-	+
Availability	+	-	+	+	+-
Version Visibility	++	+-	+-	++	+-
Multiple Levels of Abstraction	++	++	+-	-	+
Automation	+	+	+	+-	+
Visualization	+	+	+-	+	-

Table 4.6: Requirement Fulfilment Summary

Conc. Model	Conceptual Model of Components and Their Relationships
Rule-based	Rule-based Component Selection
CBR	Component Selection Based on Existing Configurations
No Rec.	No Reconfiguration
Patches	Reconfiguration through Patches
++	Supported
+	Partly Supported
-	Unsupported
+-	Undefined

From table 4.6 it can be seen that the modeling technique of having a conceptual model of components and their relationships has all the requirements fulfilled

completely or partially. No other technique can offer better support for any of the partially supported requirements. In order to offer better support for the partially supported requirements in the first modeling technique, the ideas in the other models with some support to these requirements can be applied to create better support for these requirements. In the requirements where partial support is due to lacking implementation, the required implementation work needs to be done.

4.6.2 Requirement Priorities

The requirement priorities were defined in chapter 3. In order to combine information provided by requirement fulfilment and requirement priorities, the fulfilment must somehow be weighted using the priorities. It can be stated that if a modeling technique has some requirement as "Undefined" it is better than having it "Unsupported". An undefined requirement can be added without conflicting the existing requirement fulfilment as in unsupported it can be stated that the basic idea of the technique is in controversy with the requirement and thus adding support for the requirement can cause problems with support for other requirements. The following weight values are used for requirement fulfilment:

Supported	1
Partly Supported	0,5
Undefined	0
Unsupported	-0,5

For each technique, requirement fulfilment for each requirement is multiplied by the requirement priority and the results are added forming a total score for the technique. The technique with the highest score can be considered best suited to respond to the requirements of the scenarios. The scores are presented in table 4.7.

From table 4.7 one can see that the modeling technique of having a conceptual model of components and their relationships has the highest score, mainly because all requirements are supported at some level. It can also be noted that the technique of learning from existing configurations (CBR) scored quite well because of good support for the high priority requirements (same level of support for the four requirements with highest priority).

4.7 Conclusions based on the Technique Evaluation

The evaluation is based on the literature available about the modeling techniques and the conclusions made according to the ideas seen from the literature. As extensive reference from software industry is lacking from all of the modeling techniques, the applicability of the methods is based mainly on theory only. This results in the evaluation to be more suggestive than an absolute truth and thus in the im-

Requirement	Conc. Model	Rule-based	CBR	No Rec.	Patches
Evolution	2,5	-2,5	2,5	-2,5	-2,5
Feature/Bug Fix Traceability	2	0	4	2	4
Backward Compatibility	4	2	4	-2	2
Availability	2	-2	2	2	0
Version Visibility	3	0	0	3	0
Multiple Levels of Abstraction	3	3	0	-1,5	1,5
Automation	1	1	1	0	1
Visualization	0,5	0,5	0	0,5	-0,5
Total Score	**18**	**2**	**13,5**	**1,5**	**5,5**

Table 4.7: Requirement Fulfilment Scores

plementation much attention must be directed towards the availability of existing implementation and the effort needed to implement the missing parts.

The final scores in 4.7 are based on a weighted sum of requirement fulfilment, which is based on the author's view on how the fulfilment can be quantified. The line between "Supported" and "Partly supported" as well as "Unsupported" and "Undefined" is somewhat unclear. Adjusting the weight factor for requirement fulfilment one can have different values for the final scores, but having a positive and negative factor for concepts "supported" and "unsupported" and using the same values for requirement fulfilment, the order of the modeling techniques would be quite similar than the one received here.

Based on the modeling technique evaluation, it can be stated, that the modeling technique of having a conceptual model of components and their relationships can be selected as a basis for the 'ideal' modeling technique. Ideas from other modeling techniques (especially the CBR approach) can be used to strengthen the base technique's support for the requirements that were only partially supported. One reason for having only partial support for some of the requirements was lack of implementation or existing implementation's unsuitability for the case at hand. For the 'ideal' modeling technique, all the needed implementation work to gain reasonable support for all requirements, must be planned and executed.

4.8 Chapter Summary

This chapter evaluated five existing product configuration modeling techniques (*Conceptual Model of Components and Their Relationships*, *Rule-based Component Selection*, *Component Selection Based on Existing Configurations*, *No Reconfiguration* and *Reconfiguration through Patches*) against the requirements set in chapter 3. Based on the evaluation a suggestion of technique usage in the creation of the 'ideal' modeling technique was given. With this information, objective 3 (see section 1.3) can be considered reached.

Chapter 5

The 'Ideal' Modeling Technique

This chapter presents the 'ideal' modeling technique to respond to the requirements set by the case company's product configuration scenarios. Chapter 4 provided the basis for technique selection by evaluating existing modeling techniques. In this chapter, the best qualities of the existing techniques are combined to form the 'ideal' modeling technique. First, the way how the existing models are used and combined is explained. Then, the modeling technique is examined from the point of view of requirement fulfilment and how the lack of support for some requirements in the existing modeling techniques is covered by the 'ideal' model. After existing modeling technique usage and all the required modifications are explained, the resulting modeling technique is described. Last, the implementation requirements are discussed to form a basis for the required data from the case company's components and for the implentation itself.

5.1 Existing Modeling Technique Usage

As said in chapter 4, the modeling technique of using a conceptual model of components and their relationships is selected as a basis for the 'ideal' model. The conceptualization presented in figure 4.2 will be used to create the representation of the information available of the case company's solution. In section 4.1, the information provided by the conceptualization was divided into two categories, configuration model knowledge and configuration solution knowledge. To further clarify the information provided by the conceptualization, configuration model knowlegde is divided into two subcategories:

- **Structural knowledge**, which represents the basic structure of the component types and their part relationships.
- **Compabitility knowledge**, which consists of the constraints between different component types and their revisions.

The biggest challenge in the implementation raises from the acquisition of the data needed to form the actual configurations. The idea of learning from the past existing in the modeling technique of component selection based on existing configurations can be utilized here to some extend. For new configurations, the new technique can be used from the start, but for already existing configurations, the information needed to create the conceptualization does not necessarily exist. As these configurations exist and are fully functional, there is no reason to leave the information unused. The existing configurations could be stored as constraints between component type revisions (the component individuals which the existing configurations consist of are instances of some component type revisions). For other data concerning components and their relationships and compatibilities, the information must be provided by anyone who can have an effect on these issues. There should be some means to input new data for the technique to be used to create valid configurations. Section 5.4 handles the issues concerning the needed implementation.

5.2 Modifications for Requirement Fulfilment

This section describes the required modifications to the base model in order to fulfil the requirements as well as possible. Especially the requirements were the fulfilment level is lower that 'Supported' (see section 4.6) are dealt with, but for the supported requirements, any additional remarks are issued if needed.

Evolution

Evolution was only partly supported by the base model, mainly because of lack of implementation support for it. The extended conceptualization with evolution support was not elaborated. As evolution is clearly the most important requirement and the support for it is not certain, special care must be taken when implementing the modeling technique. As no other modeling technique provides any ideas on how evolution is handled, the conceptualization of evolution in the base model will be used in the implementation.

Feature/Bug Fix Traceability

The way bug fixes and features are linked to configurations was missing from the base model. In order to achieve this, an additional concept is adapted to the base model conceptualization from the general ontology of configuration (Timo Soininen, Juha Tiihonen, Tomi Männistö and Reijo Sulonen, 1998) from which the base model conceptualization originated. The concept to adapt is *function* which will be linked to component type (and thus to its subconcept component type revision). A function has some kind of a description to describe the feature or bug fix it provides (like bug id etc.).

Backward Compabitility

The main concepts behind the base model's support for backward compabitility are constraints and component type revision effectivity periods. These offer basis for the requirement fulfilment, but sometimes there is a need to reconfigure parts of the solution where backward compabitility is required like in case of a bug fix. Backward compatibility might cause a need to fix the same bug in two ways, one that brakes backward compatibility but fixes the bug in an intelligent way which dramatically improves the quality of new configurations and the quick-and-dirty fix, which can be reconfigured without braking backward compatibility. Patch version can be made as component type revisions to support backward compatibility.

Availability

As discussed in the evaluation of the base technique, availability is depended on the tools available. The selection whether to use the tools described in section 4.1, to implement a new tool or to extend existing tools is made in chapter 7.

Version Visibility

In order to achieve this, the path from configuration solution knowledge (knowledge about the configurations in the field) to configuration model knowlegde (expecially compatibility knowledge) should be clear and visible. In the implementation, one must be able to link a component individual to the component type version it was originated from.

Multiple Levels of Abstraction

This was one of the most supported requirements in the base model. It should be sufficient to use the abstraction levels in figure 2.1 and to use the part definition concept to move from one abstraction level to another. In the implementation, there must be a way to move from a high abstraction level to a lower one and vice versa.

Automation

The main consideration in automation must be focused on calculating the configuration from the conceptualization information available. The applicability of e.g. the smodels-tool for this purpose can be considered. It is also possible to create an algoritm of one's own, but as the area of calculating such pattern is quite well researched, especially using constraint satisfaction-based approaches (Tommi Syrjänen, 1999), using existing tools could prove to be a more feasible solution.

Another issue is the automation of data gathering to be used as input to create component relationships and constraints. In order for the personnel producing such information to find it reasonable to input such data to a system, it should be as close as possible to the daily routines already performed. The use of existing systems where such data is inputted, like bug reporting systems, should be utilized as much as possible.

Visualization

Visualization is dependent on tool support. The WeCoTin user interface uses a tree-like structure to handle the hierarchy (abstraction levels) of the component structure and this way should be considered as an option to a graph-based tool.

5.3 Resulting Modeling Technique

This section describes the 'ideal' modeling technique based on the combination and modification of existing techniques made in the previous sections.

5.3.1 Modified Conceptualization

The main changes made to the base technique focused on the conceptual model itself. The modified conceptualization with the changes presented in the previous section is illustrated in figure 5.1. The changes made to the original conceptualization are:

- The concept *function* was added and linked to *component type* to represent features and bug fixes provided by the component. A *function* has a *type* (bug fix, feature etc.) and a *description*. This concept is used to map functionality to component types.
- *Configuration model knowledge* is divided into *Structural knowledge* and *Compabitility knowledge* to clarify the meaning of the concepts.

5.3.2 The Configuration Process

The resulting configuration process is based on the configuration task presented in figure 4.1. The configuration process can be split into two cases, the initial configuration and reconfiguration. In both cases the initial conditions are different, but the process itself is quite similar. The two cases are depicted in figure 5.2.

The difference between the initial configuration and reconfiguration lies in the existence of the configuration solution knowledge of the already installed solution in the case of reconfiguration. In the case of initial configuration, all choices can be made only on the basis of the customer requirements, but in the case of reconfiguration the configuration solution knowledge of the existing configuration must first be transferred back to configuration model knowledge, thus adding new constraints to the component selection.

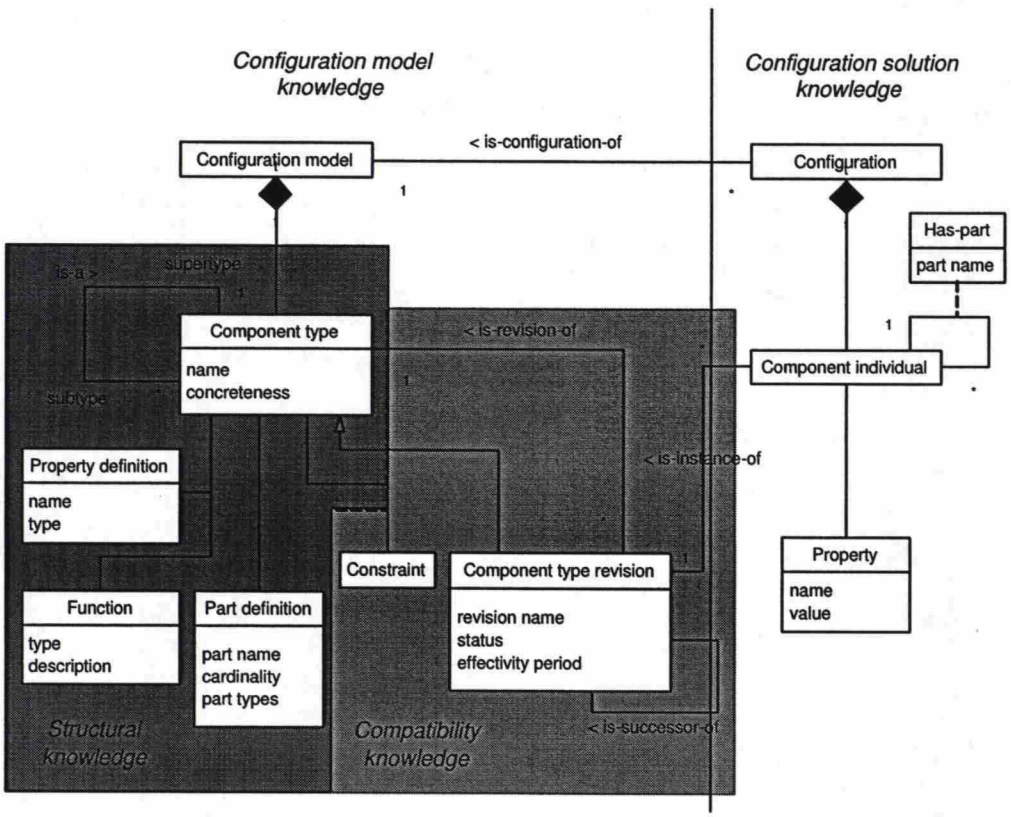

Figure 5.1: The Modified Conceptualization (adapted from (Tero Kojo, Tomi Männistö and Timo Soininen, 2003))

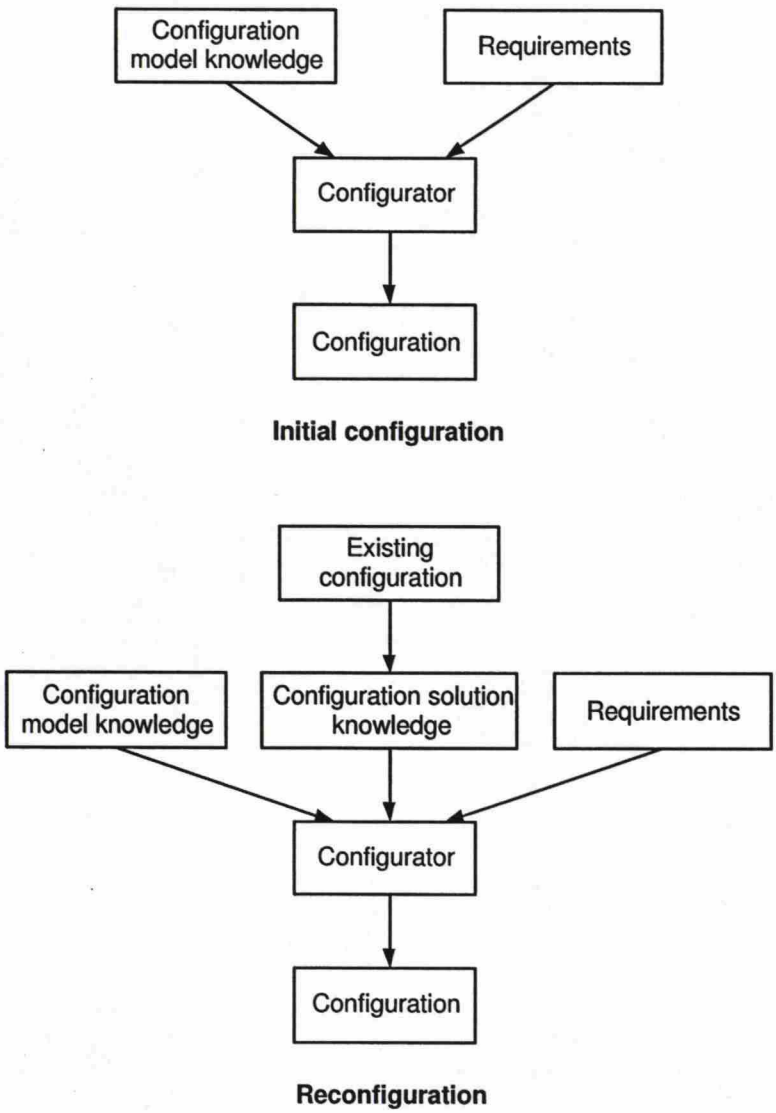

Figure 5.2: The Configuration Process

5.4 Implementation Requirements

This section summarizes the implementation requirements discussed in the earlier chapters. Most of the implementation requirements are related to some of the scenario requirements (mainly availability, automation and visualization) and some are related to the tools available and their applicability.

5.4.1 Implementing Requirement Fulfilment

Availability

The implementation requirement set by availability is that the information is available always to anyone needing it regardless of the person's location. Basically this can be done in two ways. Either the implementation is portable i.e. it can be installed to the devices which the person has with him, or it can be network-based, so that it can be accessed using a client software of a device used by the person needing the information. For the former choice, it may not be feasible to store all the information into local installations, because the amount of information will be quite large. For the latter, a WWW-based solution would be the most feasible option because it can be accessed with a basic network connection.

Automation

Because of the case company is a small mobile software company, there are little or no extra resources to spare for maintaining the modeling technique implementation. The automation should be considered in every step of the configuration process. For information gathering, it is always difficult to introduce new working habits or additional work-load, so information gathering should be based as much as possible on existing information already gathered. For requirement gathering, the user should not need to input any other data than the requirements itself. Everything else (mainly creating a configuration based on input data) should come automatically. Knowledge about existing configurations should be gathered as automatically as possible because the concequences of using obsolete data in reconfiguration may cause the creation of a non-functional configuration.

Visualization

Visualization is the requirement with lowest priority so the fulfilment of this requirement should be done as easy as possible in terms of implementation effort. This would basically mean that using a complex graph-based solution would not be feasible. Instead, using a tree-like implementation would prove to be simpler to implement and also have qualities that may prove even better for requirement fulfilment (like illustrating multiple levels of abstraction).

5.4.2 Applying Existing Tools

The main areas of implementation would be gathering model data, gathering requirements, creating configurations and gathering data about existing configurations.

Gathering Model Data

Various Product Data Management (PDM) tools exist in the market, but their support for the whole chain of implementation needed here is not known. The way the data should be presented is depended on the way the configuration model knowledge is stored. The way the case company's existing data is used and what additional data gathering systems are needed is discussed in chapter 6.

Gathering Requirements

Gathering requirements would be done using a user interface where the user requirements could be inputted in terms of features and bug fixes wanted by the user or components that are to be upgraded. One way to implement this would be an HTML-form, where the user can check different items or select from drop-down menus. The result would be displayed using a tree-like structure showing the different levels of abstraction in the resulting configuration. The WeCoTin tool presented in section 4.1 could be used here, because its use resembles the issues presented here.

Creating Configurations

Creating configurations would be the task of the *configurator*. It should implement some algoritm to combine the user requirements, possible data about existing configurations and the configuration model knowledge to form the needed configuration. Using a constraint satisfaction based implementation like *smodels* would prove useful here, but presenting the information as understandable by *smodels* would require some further investigation. The decision whether to use an existing tool or to create a new one is made in chapter 7.

Gathering Data about Existing Configurations

For reconfiguration, it is vital to have valid and up-to-date information about the existing configurations. The best way to do this is to use the case-based reasoning approach of always storing newly created configurations into a knowledge base. This would require, that before any installations are made, the configuration, which is going to be installed is created using the implementation. Also in case of upgrades the new configuration is created using the data from the existing configuration and the new configuration is stored. But in contrary to the case-based reasoning approach, the old configuration is not necessarily stored, but the old one is replaced

by the new one. If there had been no problems with the old configuration, its component composition could be used to create constraints to the configuration model but it is not needed to store the configuration as a whole.

5.5 Chapter Summary

This chapter presented the 'ideal' modeling technique, which is to be implemented. The 'ideal' modeling technique combines ideas from the existing modeling techniques evaluated in chapter 4 and modifies them where needed in order to respond as well as possible to the requirements set in chapter 3. With the description of the 'ideal' modeling technique, objective 4 (see section 1.3) can be considered reached.

Chapter 6

Information Needed from the Components

This chapter discusses the information needed from the components of the case company's solution in order for the 'ideal' modeling technique to be usable. The discussion starts by presenting existing conventions used in the case company to handle product configuration and evaluating their applicability. After evaluating the applicability of the existing conventions, the additions and modifications to these conventions is presented based on the information needed to form the conceptualization in figure 5.1. Last, a summary of the information gathered and its mapping in the conceptualization is presented.

6.1 Existing Conventions and Their Applicability

This section introduces the conventions used in the case company related to product configuration. Any ad hoc ways to handle individual cases are not discussed, only more systematic approaches.

6.1.1 Builds and Releases

Building a component out of the source files it consists of can be done in four ways in the case company:

Compile only Compiles the source files into executable files, mainly for local use on the machine where the compilation is made. This is mainly to be used by a single developer doing some initial tests with a component on his/her own machine.

Create a distribution Creates a distribution package (an archive file) out of the compiled files, which can be copied to another location to be used there. This

is used mainly to move some initial test versions of a component to a test environment, but the component is not put under official testing rounds. It is tested by a single developer or a group of developers trying out some modification they have made.

Create a build Creates a package like the one above but in addition makes a new version by tagging (Per Cederqvist et al., 2003) the source files in the version control system (CVS). A build version is an internal test version, which is put under a series of test rounds consisting of a series of test cases.

Create a release Done from a build, which it is tested to a state that it can be declared a release and delivered to customers. In some parts of the case company's products the version is described by adding the version into the name of the archive package.

Having a systematic way of keeping updated versions information available helps keep track of the evolution of the components. For the modeling technique implementation, making new builds and releases would act as input for creating new component type revisions (or patch versions) to be used in new configurations.

6.1.2 README Files

Each component contains a README file, which is in CVS. When a developer changes some parts of a component or components, he/she writes a description about the change into the README file of the corresponding component/components. When a build or release version is made out of a component, a tag is added to the README file telling what changes are included in the release. The modified README file is added to the component archive. The following shows the template used for a README entry:

```
CHANGE xxx     :  <short one-liner description here>
Change type    :  Update / New Feature / Bug Fix / Refactoring / Major Changes
Submitter      :  <submitter name>
Description    :
  <description goes here>
Note           :
  <additional notes here>
Configuration :
  <information on how to configure the change>
Usage          :
  <information on how to use this change>
Added files    :
  <added files here>
Changed files :
  <changed files here>
```

```
Removed files :
  <removed files here>
```

The entries marking added, changed and removed files are mainly needed for developer use, but other information can be used by anyone. In the one-liner description, the submitter can refer to any bug reports etc. that might be related to the change. The information on the other fields does not have an exact format and not all fields need to be filled.

The convention of documenting changes can be used in the modeling technique in many ways. It can serve as input for constraints between different components as well as provide the information needed to combine functions to component types and component type revisions. However, in order to use the information efficiently, it has to be defined more throughly in order to distinguish between constraint-related information and function-related information.

6.1.3 Creating Patch Versions

Creating patches was evaluated as a modeling technique in section 4.5. When a bug fix is needed for an older version, first the version information is gathered and then the source files for this version are fetched using the tag that was created when the component was built. Then a branch is made from this version and the changes related to the bug fix are committed to CVS into the branch and they do not affect the main trunk version. Later these changes can be merged back into the main trunk (see figure 4.5). In the case company, the branch is created by first tagging a version with a bug id and then branching from that tag.

Creating patches should not be a common practice for reconfiguration as described in section 4.5. If there is an absolute need to make a fix to a component by maintaining its original semantics (relationships to other components, functionality provided etc.) a patch version can be created.

6.1.4 Bug and Issue Tracking

In the case company, a bug tracking and issue management system called Jira (http://www.atlassian.com/software/jira/) is used. The system has concepts for component and version, which can be used for product configuration. Currently, all bugs (reported in internal testing or by customers) and new features are stored there. The system also has authentication and authorization so it could be used to achieve the requirement for availability.

6.2 Additions and Modifications Needed

This section describes the possible modifications and additions needed to the conventions in the previous section in order to be used in the modeling technique implementation.

6.2.1 Builds and Releases

The upgrade needed here is a mechanism that creates a new *component type revision* (see figure 5.1) every time a new build or release is created. The new features and bug fixes in the version are linked to it as *functions* and its relationships to other components are stored as *constraints*. Depending on the storing mechanism selected for the configuration model knowledge, the storage should be updated every time a build or release of a component is made. It can also be implemented in a way where making builds and releases is part of the user interface implementation.

6.2.2 README Files

In order to support the information update when making builds and releases, the information in the README files should be structured in a way that supports combining *constraints* and *functions* to them. The *Change type* and *Description* fields could be used to describe functions. They have a direct mapping in the conceptualization.

Handling the constraints is more complex. It may be that the implementer of a change does not necessarily have all the constraint-related information available. However, experience has proven that some constraints can already be set in this phase. At least the following changes have an effect on component interoperability and thus require some constraints to be set:

- Changing an external interface makes all components that connect to this component through the interface obsolete and they need to be changed.
- If the change is made into several components, making a new version of one component requires making new versions of all components that were changed for the change to apply.
- Adding new 3rd party components (some utility libraries etc.) requires that these components are in the configuration where the component version requiring the 3rd party component is.

Following the Linux Familiar package descriptions in (Tero Kojo, Tomi Männistö and Timo Soininen, 2003), two new fields can be added to the README file:

- **Depends** The component requires some version of another component to be installed. This can be used in the two latter cases described above.
- **Conflicts** The component cannot be used with some version(s) of another component. This can be used with the first case described above, but it requires the implementer of the change to know which components are affected by the change.

The implementer should fill in the name and version(s) of the component that the constraint concerns.

As with builds and releases, this convention could also be added as part of the modeling techniques interface. To ensure uniform syntax in component naming and versioning, the one filling in the change should be able to select the component name and version from e.g. a drop-down menu instead of typing it out to a file.

6.2.3 Creating Patch Versions

In order to ensure that the information is stored properly when a patch is made, it should be possible to complete the operations (making a branch into CVS, documenting the bug fixes etc. made to a patch) by using the user interface of the modeling technique implementation.

6.2.4 Bug and Issue Tracking

Jira was taken into use quite recently so components and versions are not yet used in a controlled and organized manner. In some parts of the products, issues and bugs are clearly connected to components and versions but not everywhere. Jira offers a good place to gather product configuration information and it is extendable. Missing functionality (mainly compability-related information) could be implemented on top of it. Some guidelines could also be created for the use of components and versions. However, the Jira structure for components and versions lacks hierarchy (components cannot have subcomponents), so in order to achieve the requirement for multiple levels of abstraction, some means of expressing hierarchy must be added.

6.3 Mapping to the Conceptualization

This section maps the information described in the previous sections into the conceptualization in figure 5.1. The mapping is depicted in table 6.1.

Convention	Conceptualization
README: Change type	Function: type
README: Description	Function: description
README: Depends	Constraint
README: Conflicts	Constraint
Build/release tag	Component type revision: revision name

Table 6.1: Mapping of Case Company Conventions to the Conceptualization

6.4 Chapter Summary

This chapter listed and described the information that is needed from the components of the case company's solution in order to implement the 'ideal' modeling technique. This information was linked to the concepts defined in the 'ideal' modeling technique in order to clarify where and how the information can be used. With this information, the model information can be formed, so objective 5 (see section 1.3) can be considered reached.

Chapter 7

Implementation of the 'Ideal' Model

This chapter describes the implementation of the 'ideal' modeling technique presented in chapter 5. The description consists of presenting the scope of the implementation in comparison with the 'ideal' model description, the technologies used and the details of the implementation. Appendix A shows how the *Synchronization* part of the case company's solution is installed at customer premices using the implementation. The example displays both the *configuration model knowledge* and *configuration solution knowledge* parts as described in the 'ideal' model conceptualization (see figure 5.1).

7.1 Implementation Scope

The implementation was decided to cover only the first two component levels of the product described in chapter 2, namely the layer level (section 2.2.1) and the subproduct level (section 2.2.2). The component level was left out of the scope of the implementation because since the summer of 2003, the scope of the company had shifted from Java-based components to C++ components with different component level structure. In order for the implementation to cope with the new product structure, it was decided to keep the component abstraction quite high to increase applicability in all parts of the case company's product offering.

As the level of abstraction was decided to be quite high, all the information presented in chapter 6 was not included in the implementation. The README files are not as important in the new product structure as they were before, so they are not included in the implementation.

In terms of releases, the focus has changed to making different kinds of releases, namely alpha release for internal use and limited usage outside the company, beta release for more extended use outside the company and then the actual release for

sales purposes.

7.2 Technologies Used in Implementation

As described in chapter 6, the implementation was done as an add-on to the existing bug and issue tracking system used in the company. The issue tracking provided the basis for the implementation, offering a place to store the information from components and component revisions. It also provides authentication and authorization.

The bug tracking system is web-based, so the implementation was done using mainly web-based technologies. The technologies are described briefly here.

XML was used to store the component structure and relationships. It provides the necessary expression to represent the 'ideal model'.

XSL was used to transform the model stored in XML into a representation that can be used by the actors in the scenarios, namely HTML.

Javascript was used to create the menu-like tree structure needed to use the information in a user-friendly manner.

PHP was used to create a level of administrational functionalities allowing the authorized persons to modify the information provided by the XML structure and make the changes apply in future references to the model.

7.3 Implementation Details

This section presents the implementation details. This consists of the parts of the bug and issue tracking system (Jira) that were used and the parts that were added in the implementation.

7.3.1 Use of Jira

As described in chapter 6, Jira offers the concepts of component and versions. By using Jira, one can link bugs and issues (new features, suggestions etc.) into certain versions and components. All components and versions are stored under one 'Project', which represents the Duality product. The view of Jira showing the organization of components and versions under a Project can be seen from figure 7.1.

Under 'Components' there are two product components, 'Browsing' and 'Synchronization' and the first has one issue and the second has two issues. Under 'Versions' there are three versions, namely '3.0.0 Synchronization', '3.0.0 Browsing' and '3.0.1 Synchronization' with each having one open issue. From this view, by

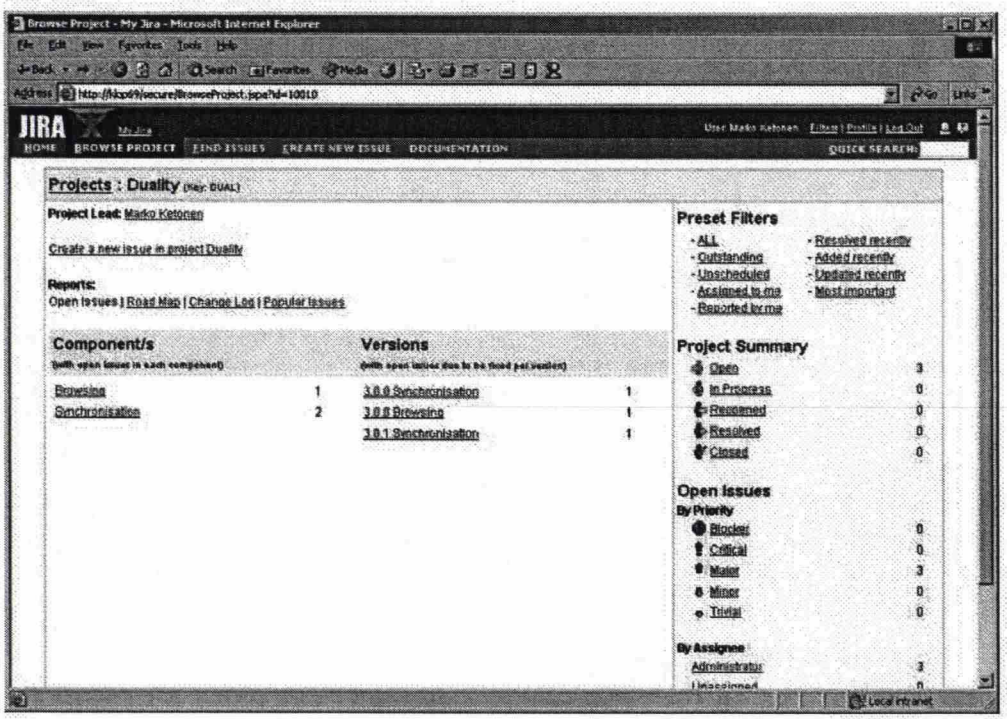

Figure 7.1: A view of Jira, components and versions under a Project

clicking one of the links under either 'Components' or 'Versions' the user is directed to a page displaying the open issues for the corresponding component and version. This view can be seen from figure 7.2. It shows two open issues for component 'Synchronisation'.

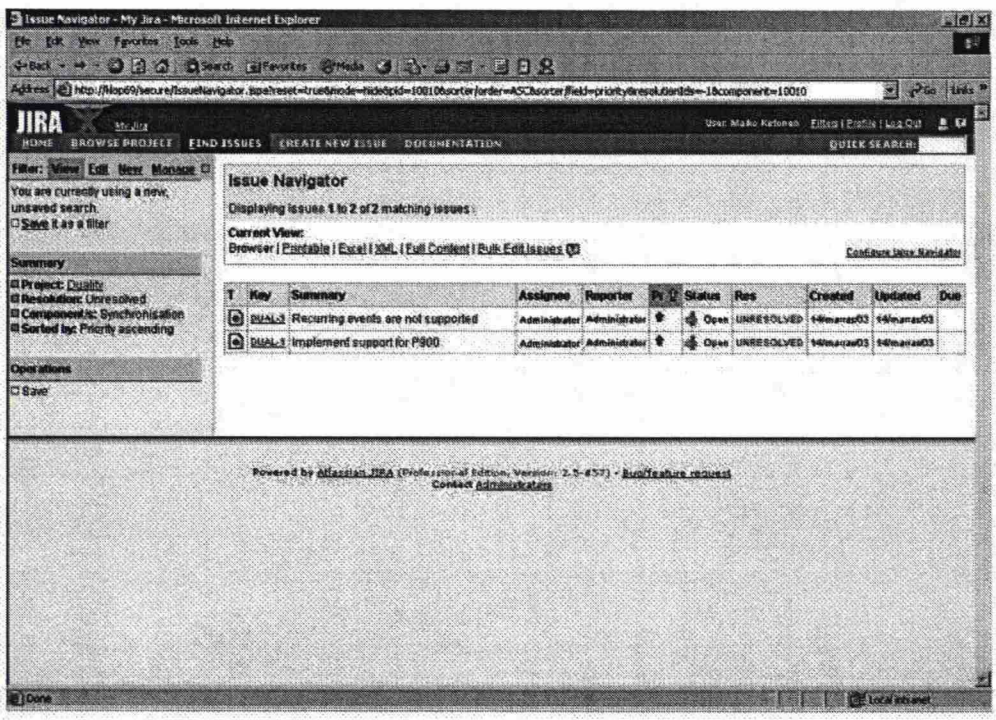

Figure 7.2: Jira open issues view

Jira offers the link from component types and component type revisions to functions (Jira issues). However, it does not offer any way of combining components as structures or combining versions to components. This functionality was added in the implementation. From the Project view, the user can only browse open issues linked to a component or version. The implementation links all issues to components and versions. This allows the user to query both issues existing in a component or version as well as issues that have been fixed for a component or version. The user can use the filtering mechanisms in Jira to sort the issues in a desired manner (e.g. using the Resolution field as sorting criteria).

7.3.2 Storing Information about Components and their Relationships

The information about components and their relationships was stored in an XML file which consists of two parts, the component types and component type revisions listed as XML elements and a tree structure containing the relationships between the components. The element list contains information about the properties linked to the component types and revisions like the links back to the issues, any documentation linked to them and a possible link to a binary distribution package if such is available. A component type or component type revision is described using the *node* XML element:

```
<node name="Duality"
      type="component"
      issuelink="/secure/BrowseProject.jspa?id=10010"
      doclink="documentation"
      binlink="binaryfile.bin"/>
```

where the following attributes can be used:

name represents the name of the component type or revision. This is also a key which is used to link the component type or revision to the tree structure of the components.

type represents the type of the node. This is used in the transformation but it also separates components from revisions.

issuelink is a link to the issue list of this component or revision in Jira as shown in figure 7.2, but shows all issues linked to the component or revision.

doclink is a link to any documentation existing about this component or revision.

binlink is a link to a binary distribution of this component or revision.

The *node* elements can be structured into a tree using *noderef* elements. The *noderef* element can contain an additional *env* attribute which is used if the relationship exists only in certain conditions (with certain revisions, customer environments etc.). The *nameref* attribute links an element to a *node* in the node list. The following shows an example of the 'Browsing' component:

```
<noderef nameref="Browsing">
  <noderef nameref="3.0.0 Browsing"/>
</noderef>
```

In the example, the *noderef* element with *nameref* 'Browsing' (links it to the 'Browsing' *node* in the node list) has a revision '3.0.0 Browsing' which has its own *node* in the node list.

7.3.3 Transforming the Information

The XML structure presented in the previous section offers very little information to the actors of the scenarios if presented as XML. Thus the information must be transformed into something that the actors can use as a source of information. The transformation is done using Extensible Stylesheet Language (XSL). It contains instructions how an XML document is transformed into other languages like HTML. The same web browser which is used to browse through Jira can be used to access the HTML document produced by the transformation. A tree-like structure was selected to visualize the model information. The transformation combines the node list in the XML file to the node structure.

In order to display only the information needed by the user, the tree structure was implemented in a way that the user can select the nodes which are visible and which are not. This was done by implementing a menu-like structure using Javascript. In order to access the leaf nodes or intermediate nodes directly, a search function was added. It opens the tree structure all the way to the nodes which match the query set by the user. Figure 7.3 shows the tree structure with some nodes expanded and search function activated.

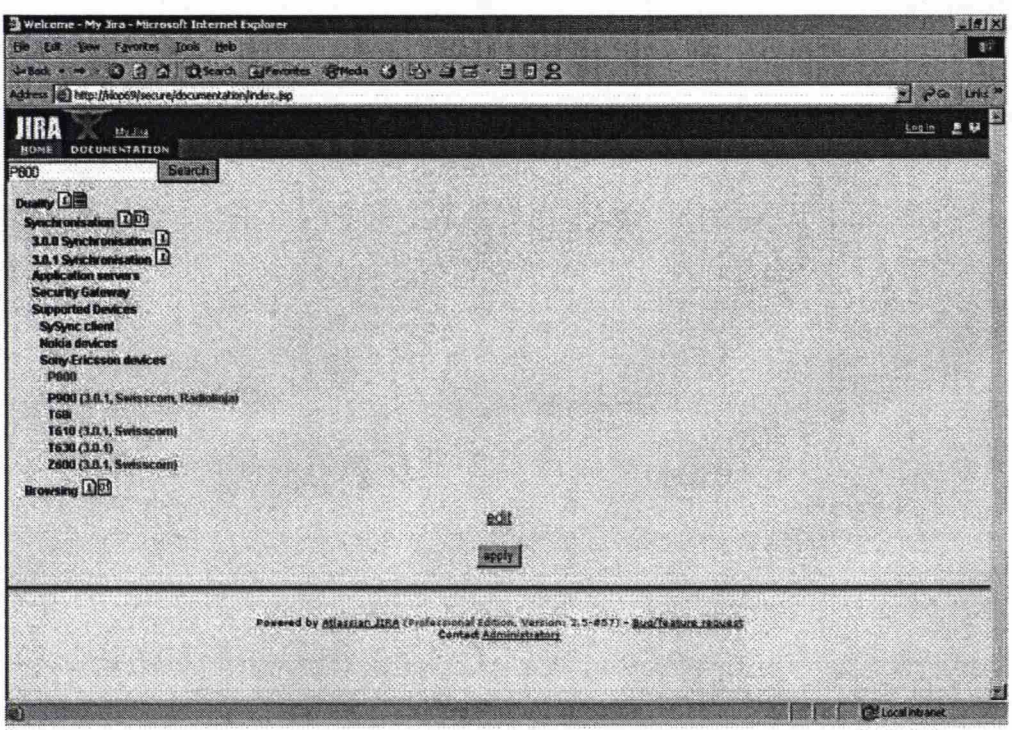

Figure 7.3: The menu-like tree structure

In the tree structure the node found by the search query (P800) is displayed in red. The icons after the node names represent the link attributes presented in section 7.3.2. They can be used to access issues, documentation and binary distributions.

7.3.4 Updating the Information

In order to keep the information up-to-date, some method of updating the structure stored in the XML file needs to exist. In figure 7.3, there are two links below the tree structure 'edit' and 'apply'. By clicking 'edit' the user receives an editor window where the XML structure can be edited. After editing and saving the file, the user can click 'apply' to make the changes visible in the tree structure. The applying was implemented using a PHP script which executes the XSL transformation.

7.3.5 Creating New Configurations

The XML structure is intended to store both the model information (component types and component type revisions) as well as the installation information consisting of the configurations installed in customer premises. When a new configuration is made, the model structure without the actual node list is copied and the root node of the new structure is named after the customer. Only the valid parts of the structure are copied, which means the components and component versions that are actually installed (two or more versions of the same component are not installed, only one). In the new configuration, new nodes (new supported devices which are tested by the customer etc.) can be added, but they must also be added to the model (as feedback for use in new configurations) by using the *env* attribute in the *noderef* element. But as the relationship of the new component is shown with the customer tag in it, the creator of a new configuration can consider this relationship with more caution than a relationship with no customer tags.

7.4 Chapter Summary

This chapter described how the 'ideal' modeling technique from chapter 5 was implemented. This consisted of describing the technologies used and how the implementation is to be used to form the product configuration modeling information needed and how the information is maintained to keep it up-to-date. The information in this chapter responds partly to objective 6 (see section 1.3), but whether the objective was reached or not can be decided in the next chapter, where the implementation is evaluated.

Chapter 8

Evaluation of the Implemented Model

In this chapter, the implementation presented in the previous chapter is evaluated against the original scenarios (see section 1.2.1) and the requirements derived from the scenarios (see chapter 3). First, the evaluation is done against the requirements (taking into account the implementation requirements set in section 5.4) and then from the point of view of the actors of the scenarios. The goals set for the scenarios in section 1.3 are also considered.

8.1 Evaluation against the Requirements

In this section, the implementation is evaluated againts the requirements that were derived from the case company's product configuration scenarios. The implementation requirements set in section 5.4 are also considered.

8.1.1 Evolution

Jira offers a possibility to create versions and components before they actually exist. This allows to create a road map of the product and by bringing the new versions and components into the XML structure as well, one can say that evolution is supported.

8.1.2 Feature/Bug Fix Traceability

As both the versions and components in Jira and the nodes in the XML tree are connected to the issues linked with them, the users can trace issues (bug fixes and features) by using the information stored in Jira and in the XML tree structure.

8.1.3 Backward Compatibility

As all old data always remains in the model stored in the XML structure, the user is always able to see the structure of old component compositions. As the information can be displayed, they can be taken into account when creating new configurations.

8.1.4 Availability

Jira offers access to the system wherever there is access to the web. With valid credentials to access Jira, a user can access the system. The implementation requirement suggested a WWW-based solution, which was implemented.

8.1.5 Version Visibility

As the whole structure in terms of components and their versions is stored, the information is available. But it requires that whenever a configuration is upgraded, the information both in Jira and in the model is upgraded as well.

8.1.6 Multiple Levels of Abstraction

The XML structure allows components to be structured in a hierarchial manner, which offers the user multiple levels of abstraction. As all tree nodes are not displayed automatically, a user can only display the nodes from which information is needed.

8.1.7 Automation

Unfortunatelly, maintaining the model in terms of new configurations and changes in old ones requires some manual work to be done (mainly editing the XML file). This is acceptable because the list of people needing to upgrade the model information is quite limited and it can be taught how to do it properly. Maintaining the issue lists connected to components and versions also requires manual work. This consists mainly of copying the links from the Jira Project page (the links to components and versions) to the XML file.

The implementation requirement for automation raised two issues, gathering of information and creating new configurations. For gathering information, the implementation relies on an existing working habit in the case company (using Jira for bug reporting and issue tracking in general). The latter requirement for creating new configurations automatically cannot be considered fulfilled because it must be done manually (see section 7.3.5).

8.1.8 Visualization

The menu-like tree structure offers a comprehensive view to the information stored in the model. However, some actors might require having any node as the root (e.g. connecting supported devices to certain component versions). This would require changing the visualization to a graph-like structure, which is not easily achievable using the techniques used in the implementation. These actors can use the search option to access the tree from leafs to root. The implementation requirement for visualization suggested using a tree structure, which was implemented.

8.2 Evaluation against the Scenarios and their Actors

In this section, the implementation is evaluated against the original scenarios (see section 1.2.1) from which the requirements were derived. The evaluation is done from the point of view of the actors in the scenarios. As all the requirements did not come from all scenarios, the scenarios are also avaluated from the point of view of the requirements that were derived from a the scenario. Table 8.1 shows which requirements were derived from which scenarios. For all scenarios, the reaching of the objectives set in 1.3 is considered.

Requirement	**Set by Scenarios**
Evolution	Reconfiguration
	Supportability and Maintainability
	The Sales Perspective
	The Future Aspect
Feature/Bug Fix Traceability	Reconfiguration
	Supportability and Maintainability
	The Future Aspect
Backward Compatibility	Reconfiguration
	The Sales Perspective
	The Future Aspect
Availability	Reconfiguration
	Supportability and Maintainability
	The Sales Perspective
Version Visibility	Reconfiguration
	Supportability and Maintainability
Multiple Levels of Abstraction	Supportability and Maintainability
	The Sales Perspective
	The Future Aspect
Automation	Reconfiguration
Visualization	The Sales Perspective

Table 8.1: The Requirements Set by the Scenarios

8.2.1 Reconfiguration

Actor's Point of View

In the reconfiguration scenario, the actor is a system integrator either working for the case company or not. An existing configuration is upgraded (or a new one installed with the customer environment acting as the existing configuration) using any information available about the available new configuration options.

By using the implementation, the system integrator can view the customer installation and the available configuration options side by side. From the model information, provided with the additional functionality needed, the system integrator can either browse or search for the needed components and also access the binary distributions if they are available. The possible upgrade must be done by hand and after the installation of the new configuration, the system integrator must upgrade the model information to match the new installation. As the upgraded components were selected using the model showing the possible selections, the new installation should follow the model as well.

Requirements Point of View

Evolution

This scenario provided the *evolution* requirement because of the need to replace existing components with new versions of the component. As the tree structure and Jira provides versions as a concept, upgrading a component into a new one in a configuration can be done by replacing the *noderef* referencing to the old version by referencing to the new version.

Feature/Bug Fix Traceability

This requirement was provided, because the actor needs to know what is gained by the component upgrades. The issues related to a version can be easily viewed using the tree structure and Jira.

Backward Compatibility

As the actor can view the old version and the new version of a component side by side in terms of e.g. supported devices, knowledge if something is lost (e.g. some device is no longer supported) can be seen straight from the comparison.

Availability

Availability is very important here, because in the case of reconfiguration, it is usually done outside office. The WWW interface of the implementation can usually be accessed wherever there is an Internet connection available.

Version Visibility

If the model information is kept up-to-date, the versions of currently installed components in all customer environments can be reviewed using the implementation.

Automation

The automation requirement was raised due to the possibility for the actor to make mistakes in reconfiguration as all procedures are done by hand. With the implementation, automation is actually compromized, because the implementation causes an extra activity to be done (to upgrade the model after the reconfiguration is done). However, the manual task of gathering information about what to upgrade is made easier, because all information is stored in one place.

Achieving the Goal

The goal for this scenario (Goal 1, see section 1.3) was to limit the amount of people needing to participate in the reconfiguration process to the system integrator only. With the implementation, the system intergrator has access to all the information needed to complete the reconfiguration, so the goal is achieved.

8.2.2 Supportability and Maintainability

Actor's Point of View

For this scenario, the actor is a support person needing information to answer questions related to having bug fixes available for an existing configuration. By using the implementation, the actor can access first the configuration existing in the customer premices and then compare the configuration to possible configurations stored in the model. He/she can also access the new bug fixes or features available for the new component versions.

Requirements Point of View

Evolution

As bug fixes are usually done in one component only, this requirement was introduced in this scenario also. The support person can view the installed components against available versions of the corresponding components. However, the support person should have enough knowledge about the component composition of the product that bug fixes can be mapped to a single component. As the implementation was done only to cover down to the subproduct level, the mapping should not cause problems.

Feature/Bug Fix Traceability

This requirement was raised here mainly because of communicating issues to customers. Jira itself can show bugs by showing both the versions the bug exists in as well as where it possibly is fixed. By using information provided by this in comparison with the installation information provided by the model, the support person can communicate possibilities for installing a bug fix to a customer.

Availability

As Jira was used by the support personnel before, there should be no additional effort in learning to use the additional implementation.

Version Visibility

Version visiblity was raised because the support person needs to know what versions of the components the customers are talking about. The implementation offers a place to store all the configurations, where they can be accessed when needed.

Multiple Levels of Abstraction

As the support person communicates issues directly to the customer, this requirement was raised. The tree structure offers a chance to look at the component composition of the configurations at any level thus the support person can choose the level of abstraction.

Achieving the Goal

The goal for this scenario (Goal 2, see section 1.3) was to reduce latency in answering customer questions about configurations and to reduce the amount of people needed to participate in the resolution to the support person only. As the implementation provides an efficient and visual way of looking into the product structure, the answer can be found quickly. The existing configurations are in the tree model as well as the available options for improvement. As in the reconfiguration scenario, one person can access and use the information. The goal is achieved.

8.2.3 The Sales Perspective

Actor's Point of View

In this scenario, the actor is a sales representative needing to answer product-related questions asked by a potential customer. By using the implementation, the actor can access information about supported devices and platforms by searching or browsing the tree structure.

Requirements Point of View

Evolution

Evolution requirement is linked to the sales perspective mainly because of the environment surrounding the product (end user devices, mail servers etc.). The model can store the environment as components and as mentioned before, they can also be versioned.

Backward Compatibility

For the case of after-sales, the sales representative can view the existing configuration side by side with the available options. Thus the decision on what can be sold can be done using the implementation.

Availability

The sales negotiations usually take place outside the office, so the sales representative needs to be able to use the information anywhere. This can be done by having a WWW access available.

Multiple Levels of Abstraction

As with the *Supportability and Maintainability* scenario, the communication to customers is a key issue. As the customer representatives usually have even lower level of technical expertise as the ones the support engineer is in contact with, the information needs to be available at a very high level. The tree structure makes this possible.

Visualization

Visualization exists only in this scenario. For sales purposes, there should be a representative product structure available. The tree structure provides one kind of visualization and it can be decided by the sales representative whether to show this to a customer or not.

Achieving the Goal

For this scenario, the goal (Goal 3, see section 1.3) was to give a sales representative some means of describing the product structure and a way to ask the right questions in terms of environment from the customer. The tree structure can be used to describe the product structure and the environment is stored with the model, so the goal is achieved.

8.2.4 The Future Aspect

Actor's Point of View

As new versions and components can be stored into Jira before they actually exist, the actor of this scenario can do so and also link new features to these future releases. By also adding these versions into the tree structure, people needing information about possible future product configurations can access this data using the implementation.

Requirements Point of View

Evolution

Evolution is in a key role in this scenario. As new features are added by adding new components, they are also added modifying the old ones. And as we look at the product at a high level, new features usually affect one high level component. By using Jira, the features can be linked to both low level and high level components. Also creating versions before they exist is possible using the knowledge from the company road map.

Feature/Bug Fix Traceability

By using the implementation, the actors of this scenario can communicate the company road map to other people (like sales representatives) in terms of product components. The changes in environment (e.g. new supported devices) can also be documented.

Backward Compatibility

The actors in this scenario can use the model to gain information about the components and versions that are present in existing configurations. By having this comprehensive view on the possible backward compatibility issues, future planning can take into account the existing configuration options that need to be preserved.

Multiple Levels of Abstraction

This requirement was raised in this scenario because of the need of expressing and linking new features into components. Jira offers a chance to create new features and link them into existing or non-existing product components and versions.

Achieving the Goal

The goal for this scenario (Goal 4, see section 1.3) was to provide means of describing non-existing configurations based on the company road map and customer

expectations. The implementation provides the means to do this, so the goal is achieved.

8.3 Evaluation Summary

Evaluation against both the requirements and the scenarions where they were derived from proved that the implementation serves its intended purpose well. It responds to the requirements and also provides means for the actors of the scenarios to improve performance in product configuration related issues. Table 8.2 summarized which implementation features support the fulfilment of the requirements.

Requirement	Set by Scenarios	Implementation Support
Evolution	Reconfiguration Supportability and Maintainability The Sales Perspective The Future Aspect	Tree structure, Jira 'Version', Jira issue view, tree view, viewing 3rd party components
Feature/Bug Fix Traceability	Reconfiguration Supportability and Maintainability The Future Aspect	Jira issue view, browsing issues from installation view
Backward Compatibility	Reconfiguration The Sales Perspective The Future Aspect	Tree view with installations and available options viewed
Availability	Reconfiguration Supportability and Maintainability The Sales Perspective	Web access
Version Visibility	Reconfiguration Supportability and Maintainability	Installations in tree structure
Multiple Levels of Abstraction	Supportability and Maintainability The Sales Perspective The Future Aspect	Only wanted nodes shown
Automation	Reconfiguration	-
Visualization	The Sales Perspective	Tree structure

Table 8.2: Summary of Requirement Fulfilment by the Implementation

Maintaining the data stored in the XML structure was not implemented in a very user oriented manner so anyone who wants to upgrade the model information needs to have information about the semantics of the XML structure. This can be

arranged through education, but a future improvement task would be to increase automation and improve the administrative interface of the implementation in order to improve the maintainability of the implementation. But for now, as the amount of people needing to make modifications into the model is limited, the current way of updating is acceptable. Implementing a more sophisticated way of updating would have taken the work needed for the implementation out of the scope of this thesis.

8.4 Chapter Summary

This chapter evaluated the implementation of the 'ideal' modeling technique from chapter 7 against the requirements set by the case company's product configuration scenarios (see section 1.2.1). The implementation proved to serve its purpose well except for the automation requirement. The automation requirement was given a low priority (see chapter 3), so its total fulfilment is not necessary. As the goals set in section 1.3 were all achieved and the implementation was done as described in chapter 7, objective 6 (see section 1.3) can be considered reached.

Chapter 9

Conclusions and Future Work

This chapter represents the analysis on how the research succeeded in responding to the original research problem. It also gives ideas on how the research can be continued in order to utilize the results of the research in a wider range of problems.

9.1 Conclusions

This research aimed to resolve a product configuration problem existing in the case company. The idea was to create a configuration modeling technique suitable to respond to requirements set by a range of product configuration scenarios in the case company. The plan was to combine the ideas of existing configuration modeling techniques and to adjust them in a way that the result would be suitable to the evolving software component environment where the case company operates.

The research was conducted by first extracting the requirements for the modeling techniques from the case company's product configuration scenarios. The requirements extracted were *evolution, feature/bug fix traceability, backward compatibility, availability, version visibility, multiple levels of abstraction, automation and visualization.* These requirements were used to evaluate five existing modeling techniques. The ideas from the modeling techniques that had the qualities to respond to the requirements were combined and modified to match the requirements in a best possible way. This resulted in the 'ideal' modeling technique, which was implemented as a part of the issue and bug tracking system already existing and in use in the case company.

The implemented model was introduced and evaluated against both the original scenarios from the point of view of their actors and the requirements extracted from them. The implemented model proved to serve the requirements well except for the maintainability of the model itself. When compared to the original goals and objectives for the study, the implementation succeeded in reaching all of them.

By the time the thesis was finalized, the implementation was not yet taken

into production, because Jira was upgraded just recently and it was decided not to include the add-on created yet. This was to avoid any problems caused by the upgrade to be linked into the add-on. As the use of the upgraded Jira is stabilized to the state where day-to-day actions can be conducted without problems, the add-on will be installed. The preliminary plan is to add the implementation within a couple of months. This caused the study to lack any indication of actual use of the implementation by the actors of the scenarios and the applicability of the implementation is solely based on the evaluation done by the author.

9.2 Future Work

The research was conducted to respond to the needs of a single mobile software company. This questions the applicability of the research results in other software companies. By analyzing problems in other companies and checking how well they match the problem field in the case company, additional data about the applicability of this research could be gained. The process of using scenarios describing a problem as basis for extracting requirements used to evaluate a solution could be generalized to be used for cases even outside the scope of this study.

The research evaluated five existing ways to conduct product configuration. This does not imply that there are not more modeling techniques that can be used for product configuration. Further research could be conducted towards the field of product configuration modeling in order to find more ways to conduct it. However, the techniques studied in the research represent a very high variety of ways to approach product configuration modeling so it is very likely that any new techniques found would at some level represent the ideas of the techniques evaluated in the research.

The research produced an implementation as a part of a commercially available issue tracking system. If a large amount of software companies match the problem field represented in the research, it would be useful both in an academic sense and commercially to analyze how a system which combines the product configuration functionalities of Jira and the additional features implemented as part of the research could form an effective and usable product configuration system.

As the implementation is taken into production use in the case company, additional research could be conducted on actual results in the scenario actor's day-to-day work.

Appendix A

Example Installation Using the Implementation

This appendix describes an example how the implementation is used in the case where the first installation is made at customer premices. The example is based on an actual installation done at the case company, but the real customer name is not displayed.

Creating a Configuration from the Model Information

First, the system integrator selects the right configuration from the model to be installed. The original installation will be made here with version "3.0.0 Synchronisation". The original model information, where the installation is started from, is depicted in figure A.1.

Creating the configuration is done by copying the relevant parts of the model XML (right version and all child nodes) as a new node with root node being the name of the new customer installation, in this case "Customer1" (actual name not shown). The resulting view is shown in figure A.2.

The installation was made on a different version of the application server than the model suggests (4.1.27 instead of 4.1.24) and it worked with this version as well. As the installation was tested by the customer, they reported that in addition to the devices said to be supported, also devices "P900" and "T610" seemed to work.

Updating the Model after Installation Information

With the additional information provided by the customer installation, the model information can be updated to include the new version of the application server and the new supported devices. But as there is some concern on whether these additional things work in all cases, the relationship is added with a customer installation

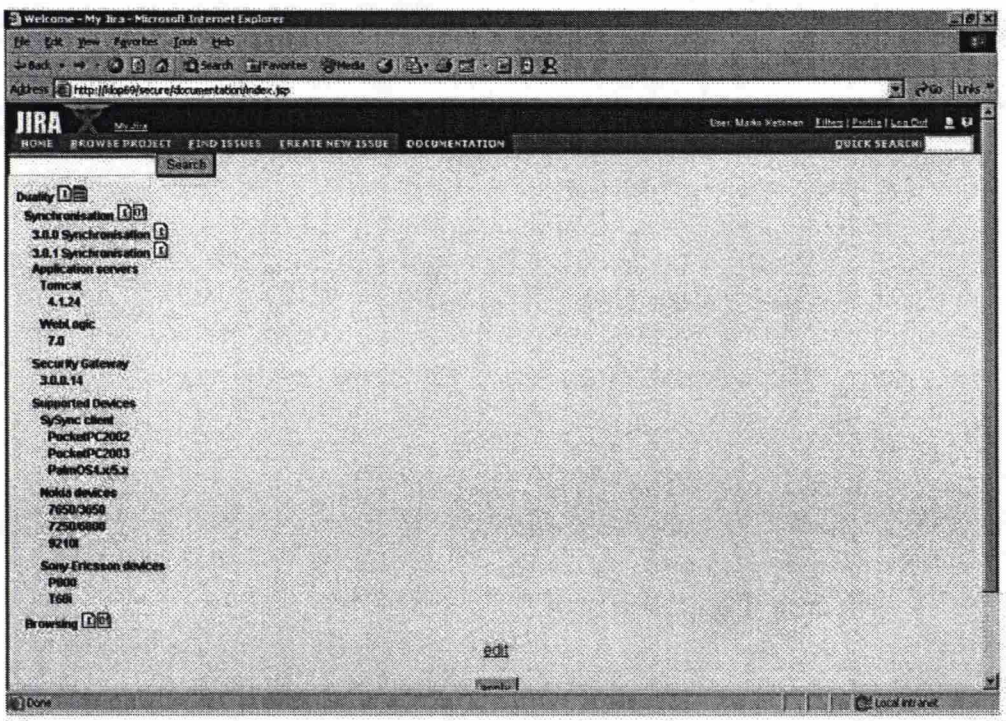

Figure A.1: The original model as basis for first installation

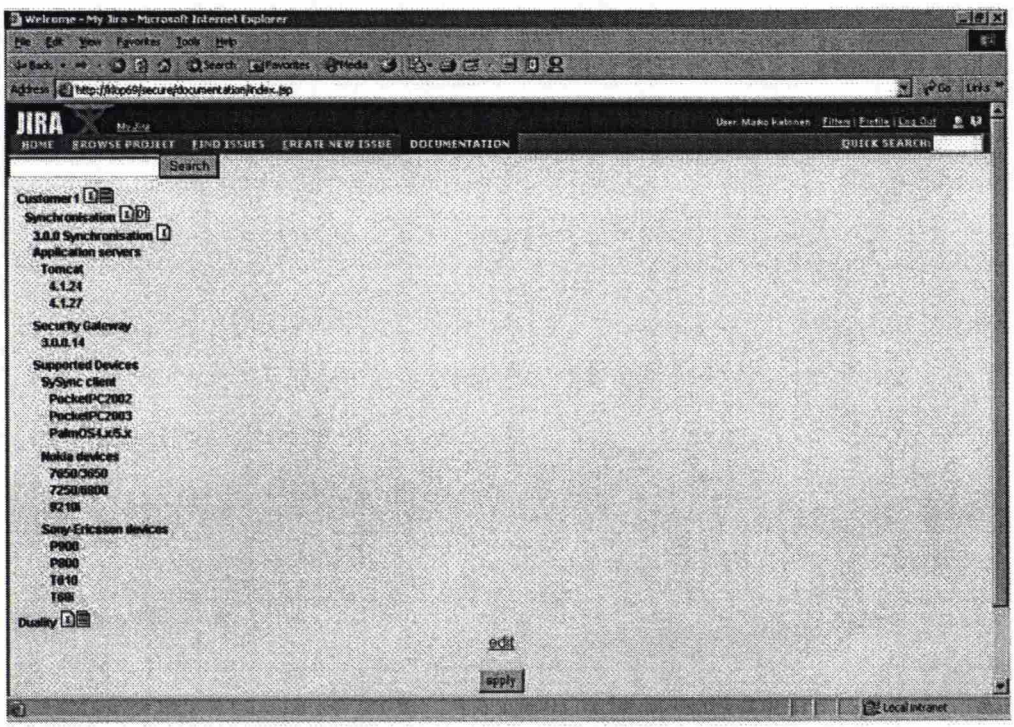

Figure A.2: The customer installation

tag. Figure A.3 shows the updated model information after the feedback from the customer installation was made.

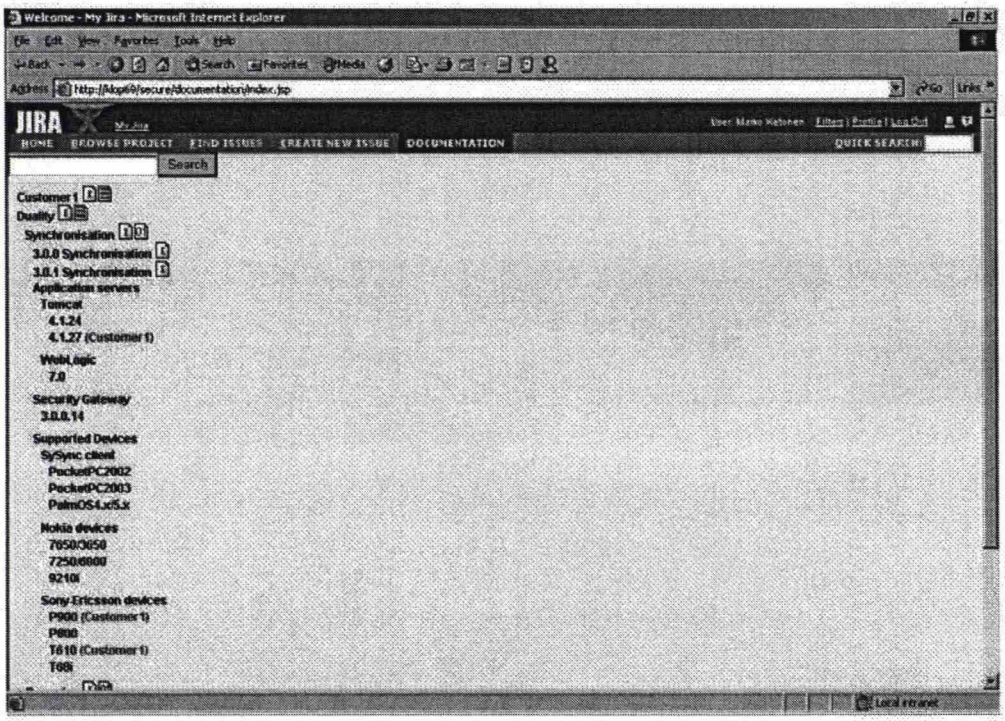

Figure A.3: The model information after feedback from installation

The figure shows how the new devices and application server version became available to the model with the "Customer1" tag in them.

Resulting XML Tree Structure

The following shows the XML structure created in the previous sections.

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="index.xsl"?>
<root>
  <nodes>
    <node name="Duality" type="component" issuelink="issues"/>
    <node name="Customer1" type="component" issuelink="issues" doclink="documentation"/>
    <node name="Synchronisation" type="component" issuelink="issues" binlink="binaries"/>
    <node name="Browsing" type="component" issuelink="issues" binlink="binaries"/>
    <node name="3.0.0 Synchronisation" type="revision" issuelink="issues"/>
    <node name="3.0.0 Browsing" type="revision" issuelink="issues"/>
    <node name="3.0.1 Synchronisation" type="revision" issuelink="issues"/>
    <node name="Application servers" type="group"/>
    <node name="Tomcat" type="component"/>
    <node name="4.1.24" type="revision"/>
    <node name="4.1.27" type="revision"/>
    <node name="WebLogic" type="component"/>
    <node name="7.0" type="revision"/>
    <node name="Databases" type="group"/>
    <node name="Oracle" type="component"/>
    <node name="8.1" type="revision"/>
    <node name="Security Gateway" type="component"/>
    <node name="3.0.0.14" type="revision"/>
    <node name="Supported clients" type="group"/>
    <node name="SySync client" type="component"/>
    <node name="PalmOS 4.x/5.x" type="revision"/>
    <node name="PocketPC2002" type="revision"/>
    <node name="PocketPC2003" type="revision"/>
    <node name="Nokia devices" type="group"/>
    <node name="Nokia 6800/7250" type="component"/>
    <node name="Nokia 3650/7650" type="component"/>
    <node name="Nokia 9210i" type="component"/>
    <node name="Nokia 6600" type="component"/>
    <node name="Sony-Ericsson devices" type="group"/>
    <node name="P800" type="component"/>
    <node name="P900" type="component"/>
    <node name="T68i" type="component"/>
    <node name="T610" type="component"/>
    <node name="T630" type="component"/>
    <node name="Z600" type="component"/>
  </nodes>
  <noderef nameref="Customer1">
    <noderef nameref="Synchronisation">
      <noderef nameref="3.0.0 Synchronisation"/>
      <noderef nameref="Application servers">
        <noderef nameref="Tomcat">
          <noderef nameref="4.1.24"/>
```

```
        <noderef nameref="4.1.27"/>
      </noderef>
    </noderef>
    <noderef nameref="Security Gateway">
      <noderef nameref="3.0.0.14"/>
    </noderef>
    <noderef nameref="Supported Devices">
      <noderef nameref="SySync client">
        <noderef nameref="PocketPC2002"/>
        <noderef nameref="PocketPC2003"/>
        <noderef nameref="PalmOS4.x/5.x"/>
      </noderef>
      <noderef nameref="Nokia devices">
        <noderef nameref="7650/3650"/>
        <noderef nameref="7250/6800"/>
        <noderef nameref="9210i"/>
      </noderef>
      <noderef nameref="Sony-Ericsson devices">
        <noderef nameref="P900"/>
        <noderef nameref="P800"/>
        <noderef nameref="T610"/>
        <noderef nameref="T68i"/>
      </noderef>
    </noderef>
  </noderef>
</noderef>
<noderef nameref="Duality">
  <noderef nameref="Synchronisation">
    <noderef nameref="3.0.0 Synchronisation"/>
    <noderef nameref="3.0.1 Synchronisation"/>
    <noderef nameref="Application servers">
      <noderef nameref="Tomcat">
        <noderef nameref="4.1.24"/>
        <noderef nameref="4.1.27" env="Customer1"/>
      </noderef>
      <noderef nameref="WebLogic">
        <noderef nameref="7.0"/>
      </noderef>
    </noderef>
    <noderef nameref="Security Gateway">
      <noderef nameref="3.0.0.14"/>
    </noderef>
    <noderef nameref="Supported Devices">
      <noderef nameref="SySync client">
        <noderef nameref="PocketPC2002"/>
        <noderef nameref="PocketPC2003"/>
        <noderef nameref="PalmOS4.x/5.x"/>
      </noderef>
      <noderef nameref="Nokia devices">
        <noderef nameref="7650/3650"/>
```

```
          <noderef nameref="7250/6800"/>
          <noderef nameref="9210i"/>
        </noderef>
        <noderef nameref="Sony-Ericsson devices">
          <noderef nameref="P900" env="Customer1"/>
          <noderef nameref="P800"/>
          <noderef nameref="T610" env="Customer1"/>
          <noderef nameref="T68i"/>
        </noderef>
      </noderef>
    </noderef>
    <noderef nameref="Browsing">
      <noderef nameref="3.0.0 Browsing"/>
    </noderef>
  </noderef>
</root>
```

Bibliography

Andreas Gnter and Lothar Holz (1999). KONWERK – a domain independent configuration tool. *Configuration Papers from the AAAI Workshop.*

Daniel Sabin and Rainer Weigel (1998). Product Configuration Frameworks-A Survey. *IEEE Intelligent Systems and Their Applications 13(4)*, 42–49.

Detlev, H. J., C. R. Roeding, G. Purkert, and S. K. L. Lindner (1999). *Secrets of Software Success: Management Insights from 100 Software Firms Around the World.* HBS Press.

Fowler, M. and K. Scott (2000). *UML Distilled: A Bried Guide to the Standard Object Modeling Language.* Addison Wesley Professional.

Gunilla Sivard (2000). *A Generic Information Platform for Product Families.* Ph. D. thesis, Stockholm Royal Institute of Technology, Department of Production Engineering.

Ilkka Niemelä and Patrik Simons (1995). Evaluating an Algorithm for Default Reasoning. *Working Notes of the IJCAI'95 Workshop on Applications and Implementations of Nonmonotonic Reasoning Systems, Montreal, Canada,.*

Juha Tiihonen, Timo Lehtonen, Timo Soininen, Antti Pulkkinen, Reijo Sulonen and Asko Riitahuhta (1999). Modeling Configurable Product Families. In *Proceedings of the 12th International Conference on Engineering Design (ICED'99), Vol. 2,* pp. 1139–1142.

Juha Tiihonen, Timo Soininen, Ilkka Niemelä and Reijo Sulonen (2002). Empirical Testing of a Weight Constraint Rule Based Configurator. In *Proceedings of the ECAI Workshop W02 on Configuration.*

Per Cederqvist et al. (2003). *Version Management with CVS.*

Roger C Schank, A. K. and C. K. Riesbeck (1994). *Inside Case-Based Explanation.* Lawrence Erlbaum Associates.

Tero Kojo, Tomi Männistö and Timo Soininen (2003). Towards Intelligent Support for Managing Evolution of Configurable Software Product Families. *Software Configuration Management (ICSE Workshops SCM 2001 and SCM 2003 Selected Papers)*, 86–101.

Timo Asikainen (2002). Representing Software Product Line Architectures Using a Configuration Ontology. Master's thesis, Helsinki University of Technology, Department of Industrial Engineering and Management.

Timo Soininen (2000). *An Approach to Knowledge Representation and Reasoning for Product Configuration Tasks.* Ph. D. thesis, Helsinki University of Technology, Department of Computer Science and Engineering.

Timo Soininen and Ilkka Niemelä (1999). Developing a Declarative Rule Language for Applications in Product Configuration. In *Practical Aspects of Declarative Languages First International Workshop, PADL'99*, pp. 305–319.

Timo Soininen, Juha Tiihonen, Tomi Männistö and Reijo Sulonen (1998). Towards a General Ontology of Configuration. *AI EDAM 12(4)*, 357–372.

Tomi Männistö, Hannu Peltonen, Timo Soininen and Reijo Sulonen (1998). Multiple Abstraction Levels in Modelling Product Structures. *Data and Knowledge Engineering 36*, 357–372.

Tomi Männistö, Timo Soininen, Juha Tiihonen and Reijo Sulonen (1999). Framework and Conceptual Model for Reconfiguration. *Configuration Papers from the AAAI Workshop*, 59–64.

Tommi Syrjänen (1998, October). Implementation of local grounding for logic programs with stable model semantics. Technical report, Helsinki University of Technology, Helsinki, Finland. Technical Report B 18.

Tommi Syrjänen (1999, December). A Rule-based Formal Model for Software Configuration. Technical report, Helsinki University of Technology, Laboratory of Theoretical Computer Science. Research Report A 55.

U. John and U. Geske (1999a). ConBaCon – Constraint Based Configuration. *Configuration Papers from the AAAI Workshop.*

U. John and U. Geske (1999b). Reconfiguration of Technical Products Using ConBaCon. *Configuration Papers from the AAAI Workshop.*